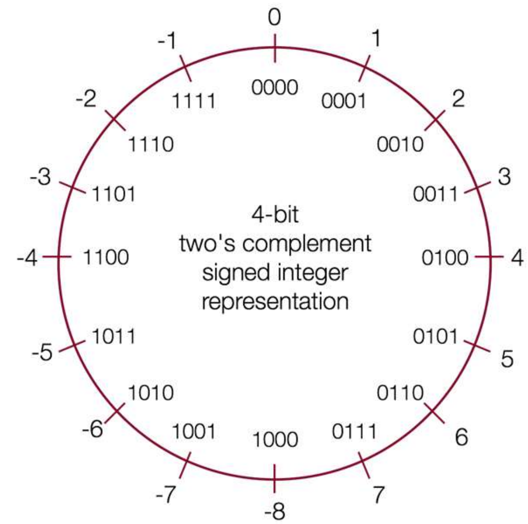
CS 107
Lecture 2: Integer Representations and Bits / Bytes

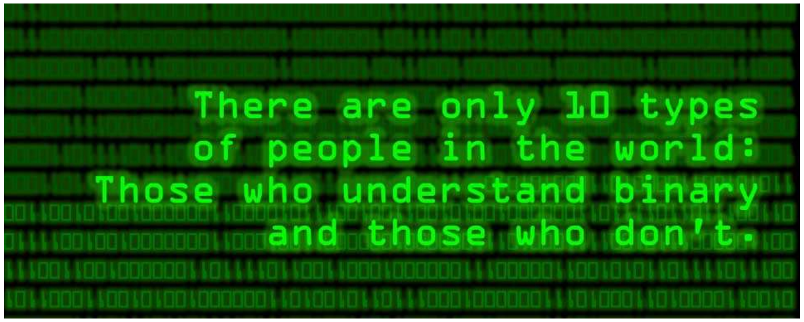
Wednesday, June 26, 2023

Computer Systems
Summer 2024
Stanford University

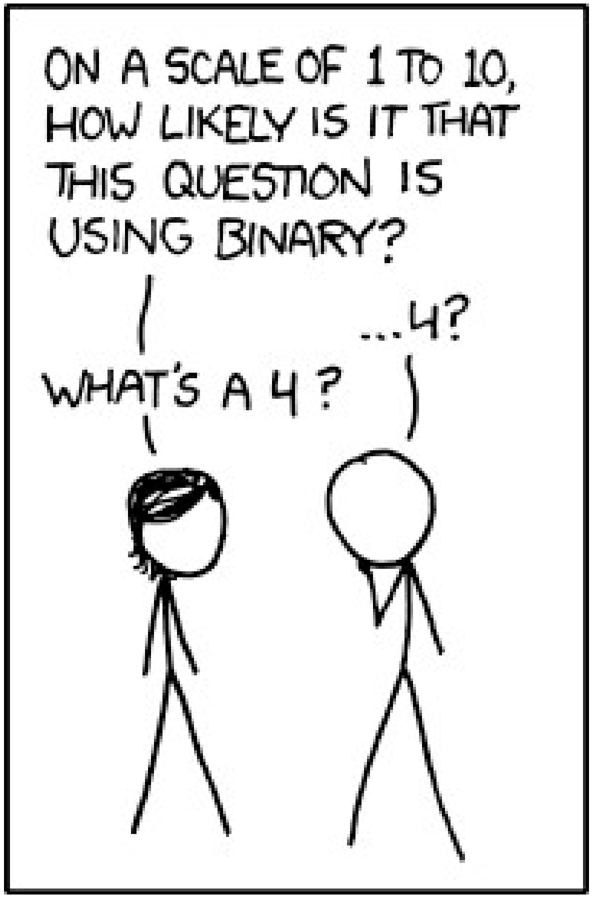
Computer Science Department

Reading:
Reader: Bits and Bytes
Textbook: Chapter 2.2





Some Binary Humor (It is Either Funny or Not)



If you get an 11/100 on a CS test, but you claim it should be counted as a 'C', they'll probably decide you deserve the upgrade. - https://xkcd.com/953/

Assignment 0: Unix!

Assignment page: https://web.stanford.edu/class/cs107/assign0/

Assignment already released, due Wednesday, 6/26

Free late days till Friday ☺

Lab

https://web.stanford.edu/class/archive/cs/cs107/cs107.1248/cgi-bin/lab_preferences

Labs will begin Week 2. Please Submit Preferences by Friday!

Today's Topics

- Numerical Bases
- Binary, Bits, & Bytes
- Octal & Hexadecimal Bases
- ASCII & Characters

- Integer Representations
  - Unsigned Numbers
  - Signed Numbers
    - Two’s Complement
    - Two’s Complement Overflow
  - Signed vs Unsigned Number Casting in C
  - Signed and Unsigned Comparisons

- Data Sizes & The `sizeof` Operator
- Min and Max Integer Values
- Truncating Integers
- More on Extending the Bit representation of Numbers
- Addressing and Byte Ordering
- Boolean Algebra

Combinations of bits can Encode Anything

We can encode anything we want with bits. E.g., the ASCII character set.

ASCII Code: Character to Binary

0	0011 0000	O	0100 1111	m	0110 1101
1	0011 0001	P	0101 0000	n	0110 1110
2	0011 0010	Q	0101 0001	o	0110 1111
3	0011 0011	R	0101 0010	p	0111 0000
4	0011 0100	S	0101 0011	q	0111 0001
5	0011 0101	T	0101 0100	r	0111 0010
6	0011 0110	U	0101 0101	s	0111 0011
7	0011 0111	V	0101 0110	t	0111 0100
8	0011 1000	W	0101 0111	u	0111 0101
9	0011 1001	X	0101 1000	v	0111 0110
A	0100 0001	Y	0101 1001	w	0111 0111
B	0100 0010	Z	0101 1010	x	0111 1000
C	0100 0011	a	0110 0001	y	0111 1001
D	0100 0100	b	0110 0010	z	0111 1010
E	0100 0101	c	0110 0011	.	0010 1110
F	0100 0110	d	0110 0100	,	0010 0111
G	0100 0111	e	0110 0101	:	0011 1010
H	0100 1000	f	0110 0110	;	0011 1011
I	0100 1001	g	0110 0111	?	0011 1111
J	0100 1010	h	0110 1000	!	0010 0001
K	0100 1011	I	0110 1001	'	0010 1100
L	0100 1100	j	0110 1010	"	0010 0010
M	0100 1101	k	0110 1011	(	0010 1000
N	0100 1110	l	0110 1100	)	0010 1001
				space	0010 0000

Number Representations

- **Unsigned Integers**: positive and 0 integers. (e.g. 0, 1, 2, … 99999…
- **Signed Integers:** negative, positive and 0 integers. (e.g. …-2, -1, 0, 1,… 9999…)

- **Floating Point Numbers:** real numbers. (e,g. 0.1, -12.2, 1.5×10^{12})

Look up IEEE floating point if you're interested! Or wait till week 7 ☺ !

Data Sizes

On the myth computers (and most 64-bit computers today), the `int` representation is comprised of 32-bits, or four 8-bit bytes. NOTE: C language does not mandate sizes. To the right is Figure 2.3 from your textbook:

C declaration		Bytes	
Signed	Unsigned	32-bit	64-bit
[signed] char	unsigned char	1	1
short	unsigned short	2	2
int	unsigned	4	4
long	unsigned long	4	8
int32_t	uint32_t	4	4
int64_t	uint64_t	8	8
char *		4	8
float		4	4
double		8	8

Data Sizes

There are guarantees on the lower-bounds for type sizes, but you should expect that the myth machines will have the numbers in the 64-bit column.

C declaration		Bytes	
Signed	Unsigned	32-bit	64-bit
[signed] char	unsigned char	1	1
short	unsigned short	2	2
int	unsigned	4	4
long	unsigned long	4	8
int32_t	uint32_t	4	4
int64_t	uint64_t	8	8
char *		4	8
float		4	4
double		8	8

Data Sizes

You can be guaranteed the sizes for `int32_t` (4 bytes) and `int64_t` (8 bytes)

C declaration		Bytes	
Signed	Unsigned	32-bit	64-bit
[signed] char	unsigned char	1	1
short	unsigned short	2	2
int	unsigned	4	4
long	unsigned long	4	8
int32_t	uint32_t	4	4
int64_t	uint64_t	8	8
char *		4	8
float		4	4
double		8	8

Data Sizes

C allows a variety of ways to order keywords to define a type. The following all have the same meaning:

```
unsigned long
unsigned long int
long unsigned
long unsigned int
```

C declaration		Bytes	
Signed	Unsigned	32-bit	64-bit
[signed] char	unsigned char	1	1
short	unsigned short	2	2
int	unsigned	4	4
long	unsigned long	4	8
int32_t	uint32_t	4	4
int64_t	uint64_t	8	8
char *		4	8
float		4	4
double		8	8

Transitioning To Larger Datatypes

- **Early 2000s:** most computers were **32-bit.** This means that pointers were **4 bytes (32 bits).**
- 32-bit pointers store a memory address from 0 to 2^{32}-1, equaling **2^{32} bytes of addressable memory**. This equals **4 Gigabytes**, meaning that 32-bit computers could have at most **4GB** of memory (RAM)!
- Because of this, computers transitioned to **64-bit.** This means that datatypes were enlarged; pointers in programs were now **64 bits.**
- 64-bit pointers store a memory address from 0 to 2^{64}-1, equaling **2^{64} bytes of addressable memory.** This equals **16 Exabytes**, meaning that 64-bit computers could have at most **1024*1024*1024*16 GB** of memory (RAM)!

Addressing and Byte Ordering

On the myth machines, pointers are 64-bits long, meaning that a program can "address" up to 2^{64} bytes of memory, because each byte is individually addressable.

This is a lot of memory! It is 16 *exa*bytes, or 1.84 x 10^{19} bytes. Older, 32-bit machines could only address 2^{32} bytes, or 4 Gigabytes.

64-bit machines can address 4 *billion* times more memory than 32-bit machines...

Machines will not need to address more than 2^{64} bytes of memory for a long, long time.

Overflow

- If you exceed the **maximum** value of your bit representation, you *wrap around* or *overflow* back to the **smallest** bit representation.

```
0b1111 + 0b1 = 0b0000
```

- If you go below the **minimum** value of your bit representation, you *wrap around* or *overflow* back to the **largest** bit representation.

```
0b0000 - 0b1 = 0b1111
```

Overflow in Unsigned Addition

When integer operations overflow in C, the runtime does not produce an error:

```
#include<stdio.h>
#include<stdlib.h>
#include<limits.h> // for UINT_MAX

int main() {
    unsigned int a = UINT_MAX;
    unsigned int b = 1;
    unsigned int c = a + b;
    printf("a = %u\n",a);
    printf("b = %u\n",b);
    printf("a + b = %u\n",c);

    return 0;
}
```

```
$ ./unsigned_overflow
a = 4294967295
b = 1
a + b = 0
```

Technically, unsigned integers in C don't overflow, they just wrap. You need to be aware of the size of your numbers. Here is one way to test if an addition will fail:

```
// for addition
#include <limits.h>
unsigned int a = <something>;
unsigned int x = <something>;
if (a > UINT_MAX - x) /* `a + x` would overflow */;
```

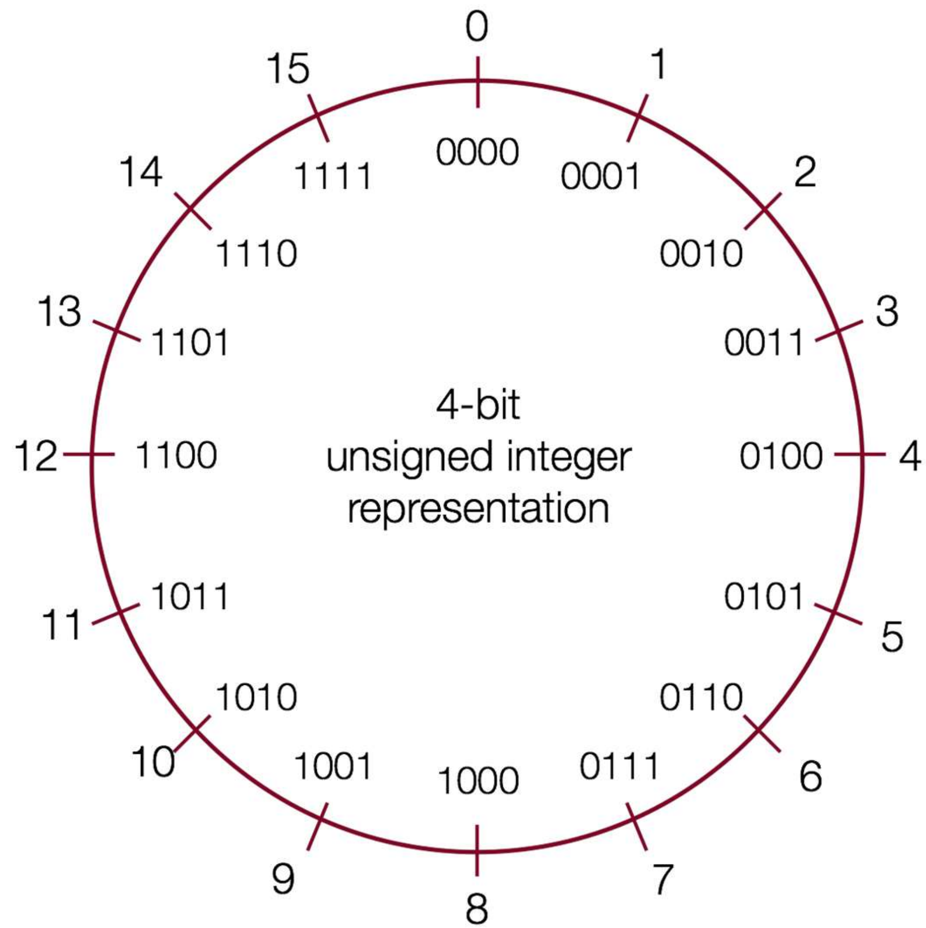
Unsigned Integers

For positive (unsigned) integers, there is a 1-to-1 relationship between the decimal representation of a number and its binary representation. If you have a 4-bit number, there are 16 possible combinations, and the unsigned numbers go from 0 to 15:

`0b0000 = 0`	`0b0001 = 1`	`0b0010 = 2`	`0b0011 = 3`
`0b0100 = 4`	`0b0101 = 5`	`0b0110 = 6`	`0b0111 = 7`
`0b1000 = 8`	`0b1001 = 9`	`0b1010 = 10`	`0b1011 = 11`
`0b1100 = 12`	`0b1101 = 13`	`0b1110 = 14`	`0b1111 = 15`

The range of an unsigned number is $0 \rightarrow 2^w - 1$, where w is the number of bits in our integer. For example, a 32-bit `int` can represent numbers from 0 to $2^{32} - 1$, or 0 to 4,294,967,295.

Unsigned Integers



Computers use a limited number of bits for numbers

```
#include<stdio.h>
#include<stdlib.h>

int main() {
    int a = 200;
    int b = 300;
    int c = 400;
    int d = 500;

    int answer = a * b * c * d;
    printf("%d\n",answer);
    return 0;
}
```

```
$ gcc -g -O0 mult-test.c -o mult-test
$ ./mult-test
-884901888
$
```

Computers use a limited number of bits for numbers

```
#include<stdio.h>
#include<stdlib.h>

int main() {
    int a =  200;
    int b =  300;
    int c =  400;
    int d =  500;
    int answer = a * b * c * d;
    printf("%d\n",answer);
    return 0;
}
```

Recall that in base 10, you can represent: 10 numbers with one digit (0 - 9),
100 numbers with two digits (00 - 99),
1000 numbers with three digits (000 - 999)

I.e., with *n* digits, you can represent up to 10^n numbers.

In base 2, you can represent:
2 numbers with one digit (0 - 1)
4 numbers with two digits (00 - 11)
8 numbers with three digits (000 - 111)

I.e., with *n* digits, you can represent up to 2^n numbers

The C `int` type is a "32-bit" number, meaning it uses 32 digits. That means we can represent up to 2^{32} numbers.

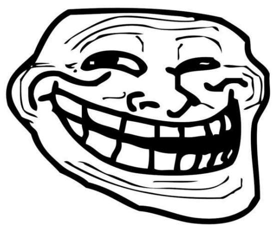
Computers use a limited number of bits for numbers

```c
#include<stdio.h>
#include<stdlib.h>

int main() {
    int a = 200;
    int b = 300;
    int c = 400;
    int d = 500;

    int answer = a * b * c * d;
    printf("%d\n",answer);
    return 0;
}
```

```
$ gcc -g -O0 mult-test.c -o mult-test
$ ./mult-test
-884901888
$
```

2^{32} = 4,294,967,296

200 * 300 * 400 * 500 = 12,000,000,000



problem?

Turns out it is worse -- `int`s are signed, meaning that the largest positive number is

$(2^{32} / 2) - 1 =$

$2^{31} - 1 = 2{,}147{,}483{,}647$

Computers use a limited number of bits for numbers

```
#include<stdio.h>
#include<stdlib.h>

int main() {
    int a = 200;
    int b = 300;
    int c = 400;
    int d = 500;

    int answer = a * b * c * d;
    printf("%d\n",answer);
    return 0;
}
```

```
$ gcc -g -O0 mult-test.c -o mult-
test
$ ./mult-test
-884901888
$
```

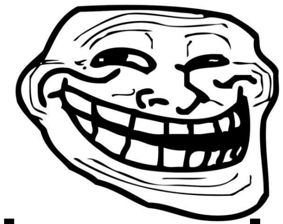
The good news: all of the following produce the same (wrong) answer:

```
 (500 * 400) * (300  * 200)
((500  *  400) *  300)  * 200
((200  *  500) *  300)  * 400
400  *  (200  *  (300  *  500))
```

Let's look at a different program

```
#include<stdio.h>
#include<stdlib.h>

int main() {
    float a = 3.14;
    float b = 1e20;

    printf("(3.14 + 1e20) - 1e20 = %f\n", (a + b) - b);
    printf("3.14 + (1e20 - 1e20) = %f\n", a + (b - b));

    return 0;
}
```

```
$ gcc -g -Og -std=gnu99 float-mult-
test.c -o float-mult-test

$ ./float-mult-test.c
(3.14 + 1e20) - 1e20 = 0.000000
3.14 + (1e20 - 1e20) = 3.140000
$
```



bigger problem!

Information Storage

Information Storage

In C, everything can be thought of as a block of 8 bits

Information Storage

In C, everything can be thought of as a block of 8 bits called a "byte"

Byte Range

Because a byte is made up of 8 bits, we can represent the range of a byte as follows:

`00000000` to `11111111`

This range is 0 to 255 in decimal.

But, neither binary nor decimal is particularly convenient to write out bytes (binary is too long, and decimal isn't numerically friendly for byte representation)

So, we use "hexadecimal," (base 16).

Hexadecimal

- When working with bits, oftentimes we have large numbers with 32 or 64 bits.
- Instead, we'll represent bits in *base-16 instead;* this is called **hexadecimal**.

$$\underbrace{0110}_{0\text{-}15}\ \underbrace{1010}_{0\text{-}15}\ \underbrace{0011}_{0\text{-}15}$$

Hexadecimal

- Hexadecimal is *base-16*, so we need digits for 1-15. How do we do this?

0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
										10	11	12	13	14	15

Hexadecimal

Hexadecimal has 16 digits, so we augment our normal 0-9 digits with six more digits: A, B, C, D, E, and F.

Figure 2.2 in the textbook shows the hex digits and their binary and decimal values:

Hex digit	0	1	2	3	4	5	6	7
Decimal value	0	1	2	3	4	5	6	7
Binary value	0000	0001	0010	0011	0100	0101	0110	0111

Hex digit	8	9	A	B	C	D	E	F
Decimal value	8	9	10	11	12	13	14	15
Binary value	1000	1001	1010	1011	1100	1101	1110	1111

Hexadecimal

- When working with bits, oftentimes we have large numbers with 32 or 64 bits.
- Instead, we'll represent bits in *base-16 instead;* this is called **hexadecimal**.

$$\underbrace{\quad 6 \quad}_{0\text{-}15}\quad\underbrace{\quad A \quad}_{0\text{-}15}\quad\underbrace{\quad 3 \quad}_{0\text{-}15}$$

Each is a base-16 digit!

Hexadecimal

- We distinguish hexadecimal numbers by prefixing them with **`0x`**, and binary numbers with **`0b`**. These prefixes also work in C
- E.g. **`0xf5`** is **`0b11110101`**

$$\mathrm{0x}\,\underbrace{\mathrm{f}}_{1111}\;\underbrace{5}_{0101}$$

Practice: Hexadecimal to Binary

What is **`0x173A`** in binary?

Hexadecimal	**1**	**7**	**3**	**A**
Binary	**0001**	**0111**	**0011**	**1010**

Practice: Hexadecimal to Binary

What is **`0b1111001010`** in hexadecimal? (*Hint: start from the right)*

`Binary`	**`11`**	**`1100`**	**`1010`**
`Hexadecimal`	**`3`**	**`C`**	**`A`**

Hexadecimal

Convert: `0b1111001010110110110011` to hexadecimal.

Binary	11	1100	1010	1101	1011	0011
Hexadecimal	3	C	A	D	B	3

(start from the **right**)

`0b1111001010110110110011`
is hexadecimal `3CADB3`

Hex digit	0	1	2	3	4	5	6	7
Decimal value	0	1	2	3	4	5	6	7
Binary value	0000	0001	0010	0011	0100	0101	0110	0111

Hex digit	8	9	A	B	C	D	E	F
Decimal value	8	9	10	11	12	13	14	15
Binary value	1000	1001	1010	1011	1100	1101	1110	1111

Hexadecimal

Convert: `0b1111001010110110110011` to hexadecimal.

Binary	11	1100	1010	1101	1011	0011
Hexadecimal	3	C	A	D	B	3

(start from the **right**)

`0b1111001010110110110011`
is hexadecimal `3CADB3`

Hex digit	0	1	2	3	4	5	6	7
Decimal value	0	1	2	3	4	5	6	7
Binary value	0000	0001	0010	0011	0100	0101	0110	0111

Hex digit	8	9	A	B	C	D	E	F
Decimal value	8	9	10	11	12	13	14	15
Binary value	1000	1001	1010	1011	1100	1101	1110	1111

Hexadecimal

Convert: `0b1111001010110110110011` to hexadecimal.

(start from the **right**)

Binary	11	1100	1010	1101	1011	0011
Hexadecimal	3	C	A	D	B	3

`0b1111001010110110110011`
is hexadecimal `3CADB3`

Hex digit	0	1	2	3	4	5	6	7
Decimal value	0	1	2	3	4	5	6	7
Binary value	0000	0001	0010	0011	0100	0101	0110	0111

Hex digit	8	9	A	B	C	D	E	F
Decimal value	8	9	10	11	12	13	14	15
Binary value	1000	1001	1010	1011	1100	1101	1110	1111

Hexadecimal

Convert: `0b1111001010110110110011` to hexadecimal.

(start from the **right**)

Binary	11	1100	1010	1101	1011	0011
Hexadecimal	3	C	A	D	B	3

`0b1111001010110110110011`
is hexadecimal `3CADB3`

Hex digit	0	1	2	3	4	5	6	7
Decimal value	0	1	2	3	4	5	6	7
Binary value	0000	0001	0010	0011	0100	0101	0110	0111

Hex digit	8	9	A	B	C	D	E	F
Decimal value	8	9	10	11	12	13	14	15
Binary value	1000	1001	1010	1011	1100	1101	1110	1111

Hexadecimal

Convert: `0b1111001010110110110011` to hexadecimal.

(start from the **right**)

Binary	11	1100	1010	1101	1011	0011
Hexadecimal	3	C	A	D	B	3

`0b1111001010110110110011`
is hexadecimal `3CADB3`

Hex digit	0	1	2	3	4	5	6	7
Decimal value	0	1	2	3	4	5	6	7
Binary value	0000	0001	0010	0011	0100	0101	0110	0111

Hex digit	8	9	A	B	C	D	E	F
Decimal value	8	9	10	11	12	13	14	15
Binary value	1000	1001	1010	1011	1100	1101	1110	1111

Hexadecimal

Convert: `0b1111001010110110110011` to hexadecimal.

(start from the **right**)

Binary	11	1100	1010	1101	1011	0011
Hexadecimal	3	C	A	D	B	3

`0b1111001010110110110011` is hexadecimal `3CADB3`

Hex digit	0	1	2	3	4	5	6	7
Decimal value	0	1	2	3	4	5	6	7
Binary value	0000	0001	0010	0011	0100	0101	0110	0111

Hex digit	8	9	A	B	C	D	E	F
Decimal value	8	9	10	11	12	13	14	15
Binary value	1000	1001	1010	1011	1100	1101	1110	1111

Hexadecim

Convert: `0b1111001010110110110011` to hexadecimal.

(start from the **right**)

Binary	11	1100	1010	1101	1011	0011
Hexadecimal	3	C	A	D	B	3

`0b1111001010110110110011`
is hexadecimal `3CADB3`

Hex digit	0	1	2	3	4	5	6	7
Decimal value	0	1	2	3	4	5	6	7
Binary value	0000	0001	0010	0011	0100	0101	0110	0111

Hex digit	8	9	A	B	C	D	E	F
Decimal value	8	9	10	11	12	13	14	15
Binary value	1000	1001	1010	1011	1100	1101	1110	1111

Decimal to Hexadecimal

To convert from decimal to hexadecimal, you need to repeatedly divide the number in question by 16, and the remainders make up the digits of the hex number:

```
314156 decimal:

314,156 / 16 = 19,634 with 12 remainder: C
19,634 / 16  = 1,227 with 2 remainder:   2
1,227 / 16   = 76 with 11 remainder:     B
76 / 16 = 4 with 12 remainder:           C
4 / 16 = 0 with 4 remainder:             4

Reading from bottom up: 0x4CB2C
```

Hexidecimal

To convert from hexadecimal to decimal, multiply each of the hexadecimal digits by the appropriate power of 16:

```
0x7AF:

7 * 16^2 + 10 * 16 + 15
= 7 * 256 + 160 + 15
= 1792 + 160 + 15 = 1967
```

Hexadecimal: It's funky but concise

- Let's take a byte (8 bits):

`165`	Base-10: Human-readable, but cannot easily interpret on/off bits
`0b10100101`	Base-2: Yes, computers use this, but not human-readable
`0xa5`	Base-16: Easy to convert to Base-2, More "portable" as a human-readable format (fun fact: a half-byte is called a nibble or nybble)

Let the computer do it!

Honestly, hex to decimal and vice versa are easy to let the computer handle. You can either use a search engine (Google does this automatically), or you can use a python one-liner:

4. cgregg@myth10: ~ (ssh)

```
cgregg@myth10:~$ python -c "print(hex(314156))"
0x4cb2c
cgregg@myth10:~$ python -c "print(0x7af)"
1967
cgregg@myth10:~$ 
```

Let the computer do it!

You can also use Python to convert to and from binary:

4. cgregg@myth10: ~ (ssh)

```
cgregg@myth10:~$ python -c "print(bin(0x173A4C))"
0b101110011101001001100
cgregg@myth10:~$ python -c "print(hex(0b1111001010110110110011))"
0x3cadb3
cgregg@myth10:~$
```

(but you should memorize this as it is easy and you will use it frequently)

Let the computer do it!

You can also use Python to convert to and from binary:

```
4. cgregg@myth10: ~ (ssh)
cgregg@myth10:~$ python -c "print(bin(0x173A4C))"
0b101110011101001001100
cgregg@myth10:~$ python -c "print(hex(0b1111001010110110110011))"
0x3cadb3
cgregg@myth10:~$
```

(also might show up in an offline exam 😉)

How to Represent A Signed Value

A **signed** integer is a negative, 0, or positive integer.

How can we represent both negative *and* positive numbers in binary?

Signed Integers

- A **signed** integer is a negative integer, 0, or a positive integer.
- *Problem:* How can we represent negative *and* positive numbers in binary?

Idea: let's reserve the *most significant bit* to store the sign.

Sign Magnitude Representation

$$\underbrace{0}_{\text{positive}}\underbrace{110}_{6}$$

$$\underbrace{1}_{\text{negative}}\underbrace{011}_{3}$$

Sign Magnitude Representation

$$\underbrace{0}_{\text{positive}}\underbrace{000}_{0}$$

$$\underbrace{1}_{\text{negative}}\underbrace{000}_{0}$$

Sign Magnitude Representation

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

- We've only represented 15 of our 16 available numbers!

Sign Magnitude Representation AKA Ones Complement

- **Pro:** easy to represent, and easy to convert to/from decimal.
- **Con:** +-0 is not intuitive
- **Con:** we lose a bit that could be used to store more numbers
- **Con:** arithmetic is tricky: we need to find the sign, then maybe subtract (borrow and carry, etc.), then maybe change the sign. This complicates the hardware support for something as fundamental as addition.

Can we do better?

Now Lets Try a Better Approach!

A Better Idea

- Ideally, binary addition would *just work* **regardless** of whether the number is positive or negative.

$$\begin{array}{r} 0101 \\ +\ ???? \\ \hline 0000 \end{array}$$

A Better Idea

- Ideally, binary addition would *just work* **regardless** of whether the number is positive or negative.

$$\begin{array}{r} 0101 \\ +\,1011 \\ \hline 0000 \end{array}$$

A Better Idea

- Ideally, binary addition would *just work* **regardless** of whether the number is positive or negative.

$$\begin{array}{r} 0011 \\ +\ ???? \\ \hline 0000 \end{array}$$

A Better Idea

- Ideally, binary addition would *just work* **regardless** of whether the number is positive or negative.

$$
\begin{array}{r}
0011 \\
+\,{\color{red}1101} \\
\hline
0000
\end{array}
$$

A Better Idea

- Ideally, binary addition would *just work* **regardless** of whether the number is positive or negative.

$$
\begin{array}{r}
0000 \\
+\,{\color{red}????} \\
\hline
0000
\end{array}
$$

A Better Idea

- Ideally, binary addition would *just work* **regardless** of whether the number is positive or negative.

$$\begin{array}{r} 0000 \\ +\,{\color{red}0000} \\ \hline 0000 \end{array}$$

A Better Idea

Decimal	Positive	Negative
0	0000	0000
1	0001	1111
2	0010	1110
3	0011	1101
4	0100	1100
5	0101	1011
6	0110	1010
7	0111	1001

Decimal	Positive	Negative
8	1000	1000
9	1001 (same as -7!)	NA
10	1010 (same as -6!)	NA
11	1011 (same as -5!)	NA
12	1100 (same as -4!)	NA
13	1101 (same as -3!)	NA
14	1110 (same as -2!)	NA
15	1111 (same as -1!)	NA

There Seems Like a Pattern Here…

$$\begin{array}{r} 0101 \\ +\ {\color{red}1011} \\ \hline 0000 \end{array} \qquad \begin{array}{r} 0011 \\ +\ {\color{red}1101} \\ \hline 0000 \end{array} \qquad \begin{array}{r} 0000 \\ +\ {\color{red}0000} \\ \hline 0000 \end{array}$$

- The negative number is the positive number **inverted, plus one!**

There Seems Like a Pattern Here...

A binary number plus its inverse is all 1s.

$$\begin{array}{r} 0101 \\ +\,1010 \\ \hline 1111 \end{array}$$

Add 1 to this to carry over all 1s and get 0!

$$\begin{array}{r} 1111 \\ +\,0001 \\ \hline 0000 \end{array}$$

Another Trick

- To find the negative equivalent of a number, work right-to-left and write down all digits *through* when you reach a 1. Then, invert the rest of the digits.

$$
\begin{array}{r}
100100 \\
+\ ?????? \\
\hline
000000
\end{array}
$$

Another Trick

- To find the negative equivalent of a number, work right-to-left and write down all digits *through* when you reach a 1. Then, invert the rest of the digits.

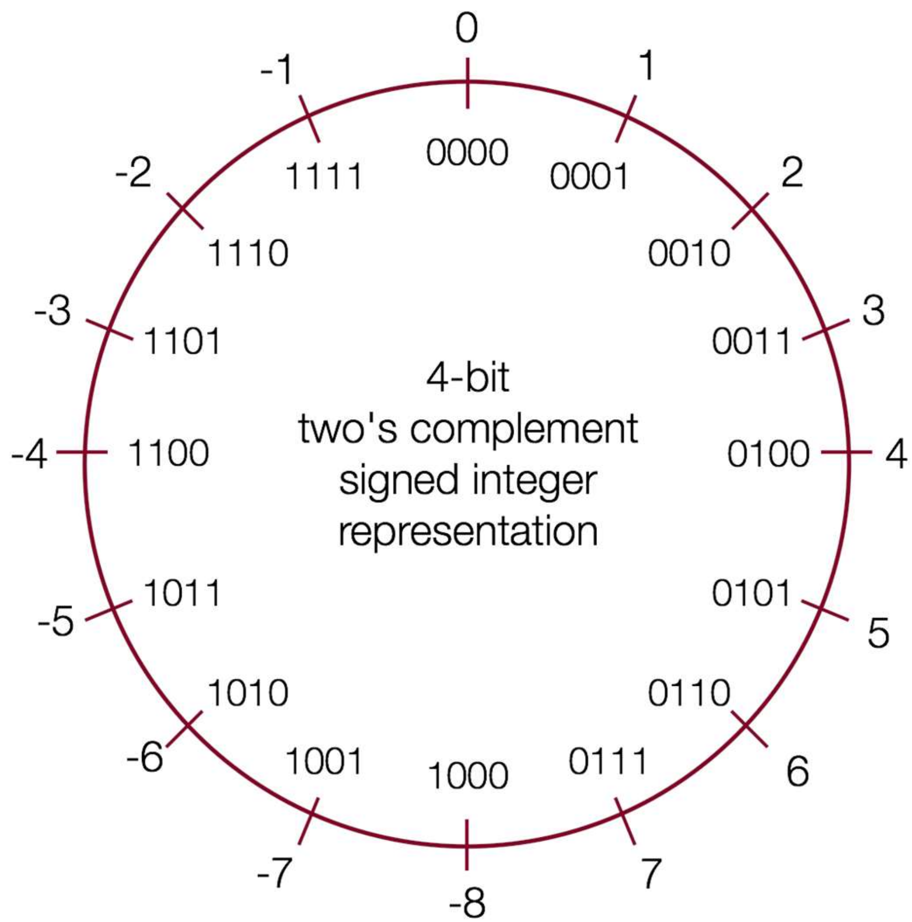
$$
\begin{array}{r}
100100 \\
+\ ???100 \\
\hline
000000
\end{array}
$$

Another Trick

- To find the negative equivalent of a number, work right-to-left and write down all digits *through* when you reach a 1. Then, invert the rest of the digits.

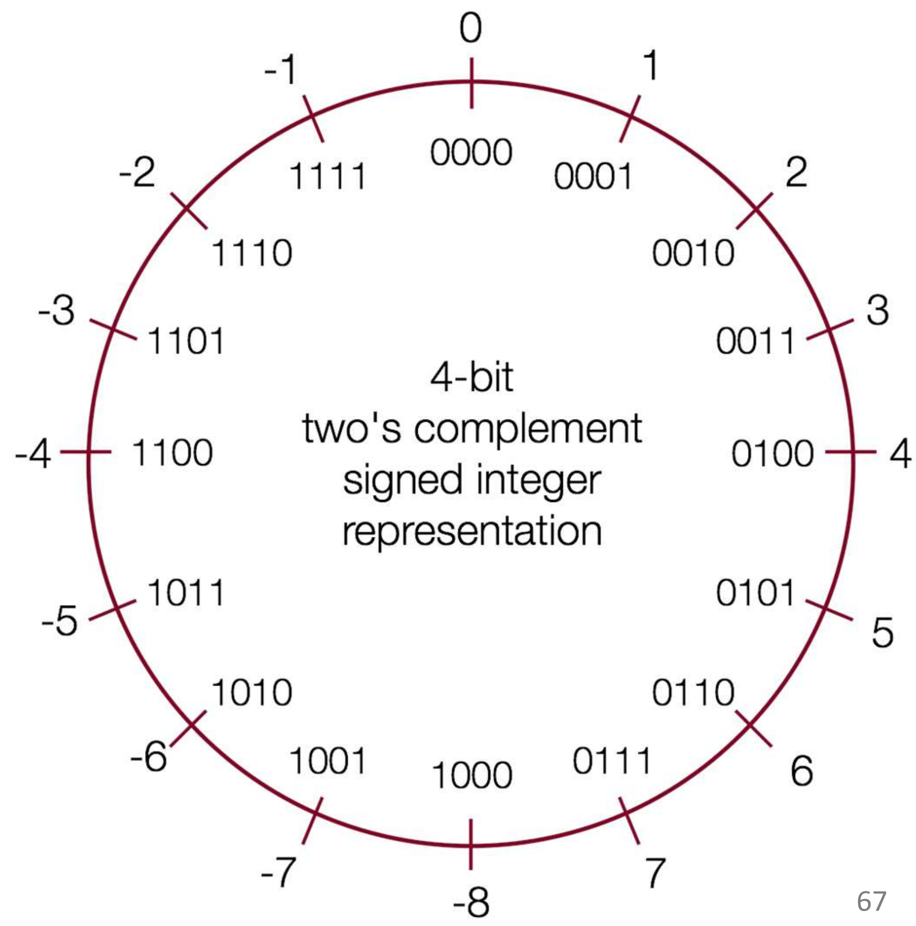
$$
\begin{array}{r}
100100 \\
+\ \underline{011100} \\
000000
\end{array}
$$

Two’s Complement



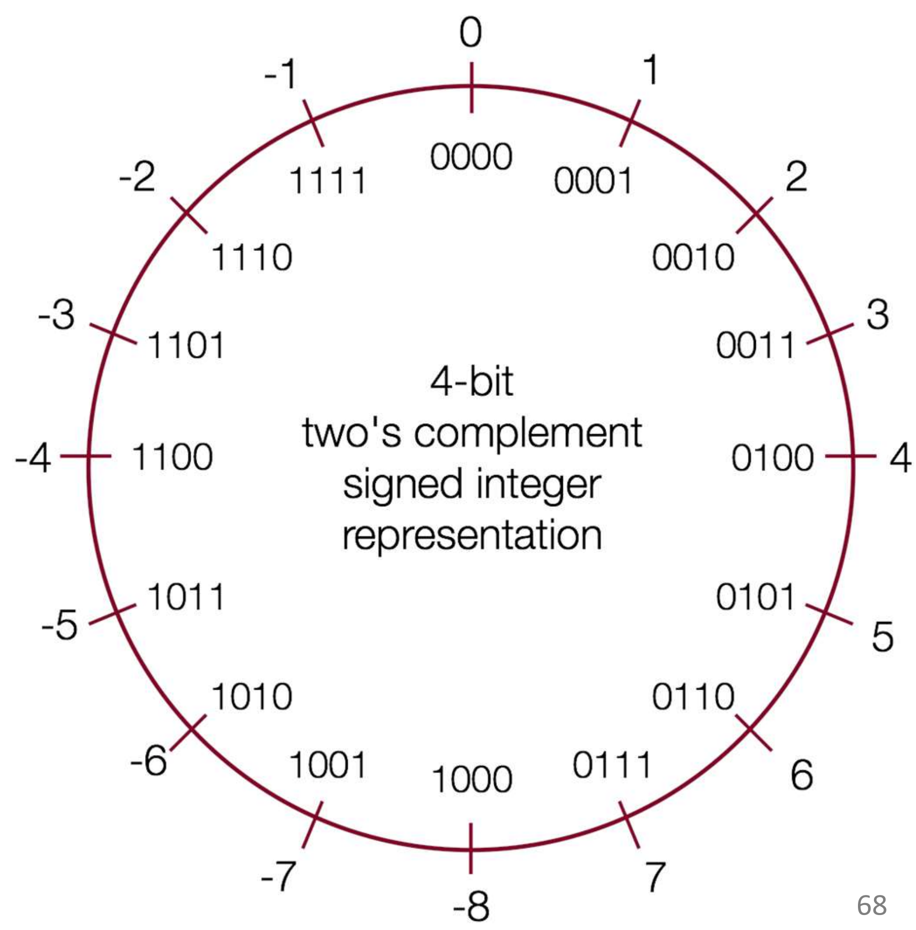
Two's Complement

- In **two's complement**, we represent a positive number as **itself**, and its negative equivalent as the **two's complement of itself.**
- The **two's complement** of a number is the binary digits inverted, plus 1.
- This works to convert from positive to negative, **and** back from negative to positive!



Two’s Complement

- **Con:** more difficult to represent, and difficult to convert to/from decimal and between positive and negative.
- **Pro:** only 1 representation for 0!
- **Pro:** all bits are used to represent as many numbers as possible
- **Pro:** the most significant bit still indicates the sign of a number.
- **Pro:** addition works for any combination of positive and negative!



Two's Complement

- Adding two numbers is just…adding! There is no special case needed for negatives. E.g. what is 2 + -5?

	0010	2
+	1011	-5
	1101	-3

Two’s Complement

- Subtracting two numbers is just performing the two’s complement on one of them and then adding. E.g. 4 – 5 = -1.

0100	4		0100	4
- 0101	5	→	+ 1011	-5
			1111	-1

How to Read Two's Complement #s

- Multiply the most significant bit by -1 and multiply all the other bits by 1 as normal

1	1	1	0
2^3	2^2	2^1	2^0

= **1***-8 + **1***4 + **1***2 + **0***1 = -2

How to Read Two's Complement #s

- Multiply the most significant bit by -1 and multiply all the other bits by 1 as normal

$$\underline{0} \quad \underline{1} \quad \underline{1} \quad \underline{0}$$

$$2^3 \quad 2^2 \quad 2^1 \quad 2^0$$

$$= \mathbf{0}*-8 + \mathbf{1}*4 + \mathbf{1}*2 + \mathbf{0}*1 = 6$$

Practice: Two's Complement

What are the negative or positive equivalents of the numbers below?

a) -4 (1100)

b) 7 (0111)

c) 3 (0011)

Go to https://pollev.com/akeppler

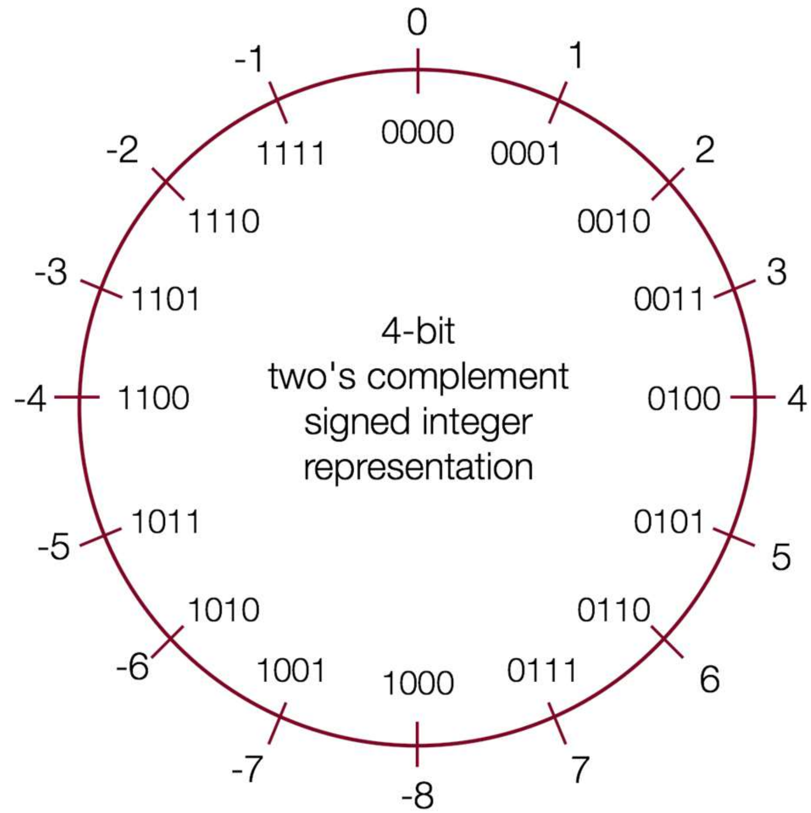
Practice: Two's Complement

What are the negative or positive equivalents of the numbers below?

a) -4 (1100) -> 4 (0100)

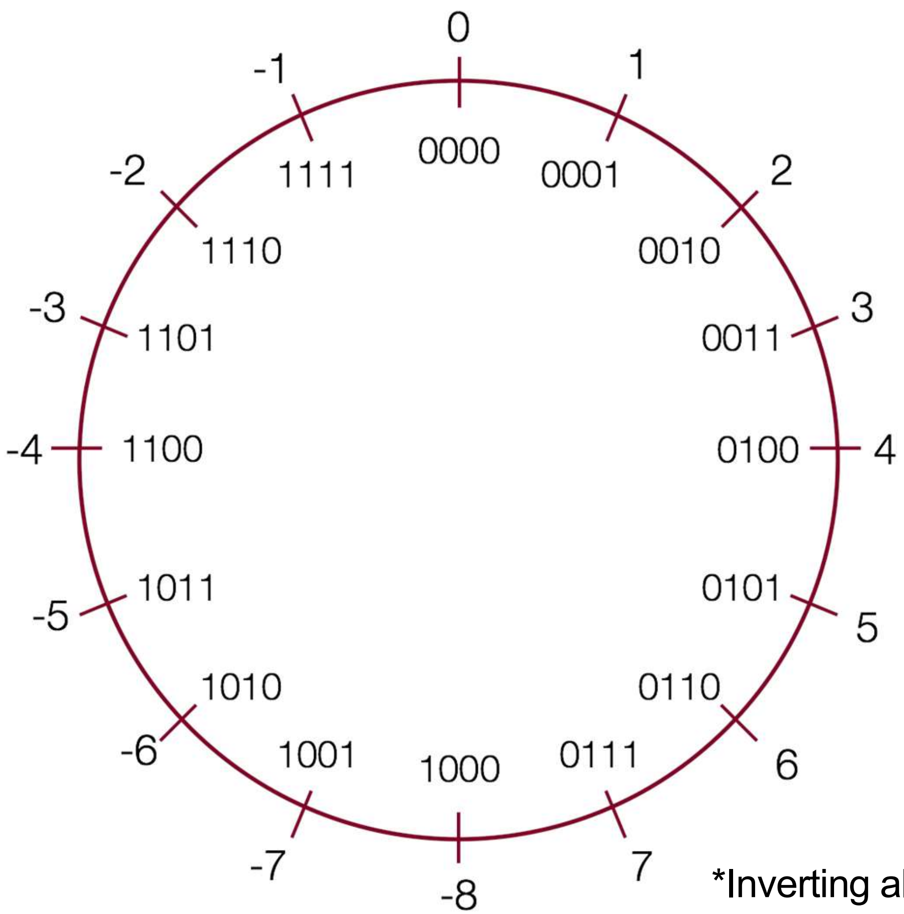
b) 7 (0111) -> (1001)

c) 3 (0011) -> (1101)



Some Extra Slides for Review

Two's Complement



In practice, a negative number in two's complement is obtained by inverting all the bits of its positive counterpart*, and then adding 1, or: `x = ~x + 1`

Example: The number 2 is represented as normal in binary: 0010

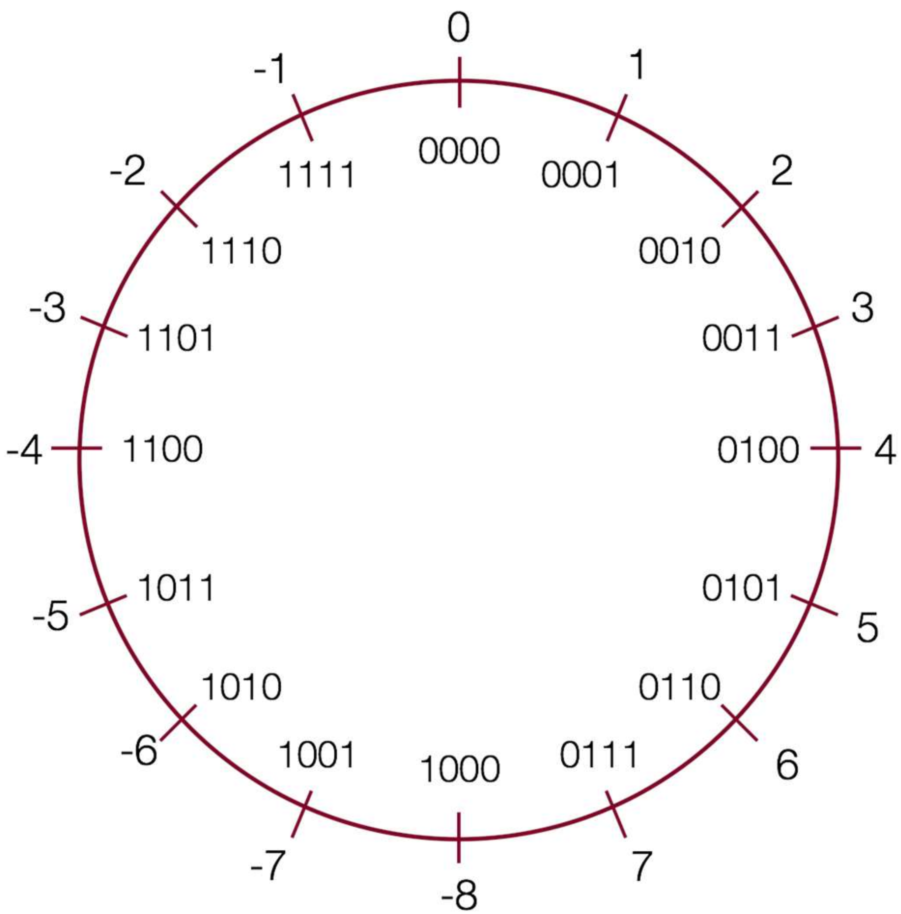
-2 is represented by inverting the bits, and adding 1:

0010 ☞ 1101

```
  1101
+    1
  1110
```

*Inverting all the bits of a number is its "one's complement"

Two's Complement



To convert a negative number to a positive number, perform the same steps!

Example: The number -5 is represented in two's complements as: 1011

5 is represented by inverting the bits, and adding 1:

1011 ☞ 0100

$$\begin{array}{r} 0100 \\ +\quad 1 \\ \hline 0101 \end{array}$$

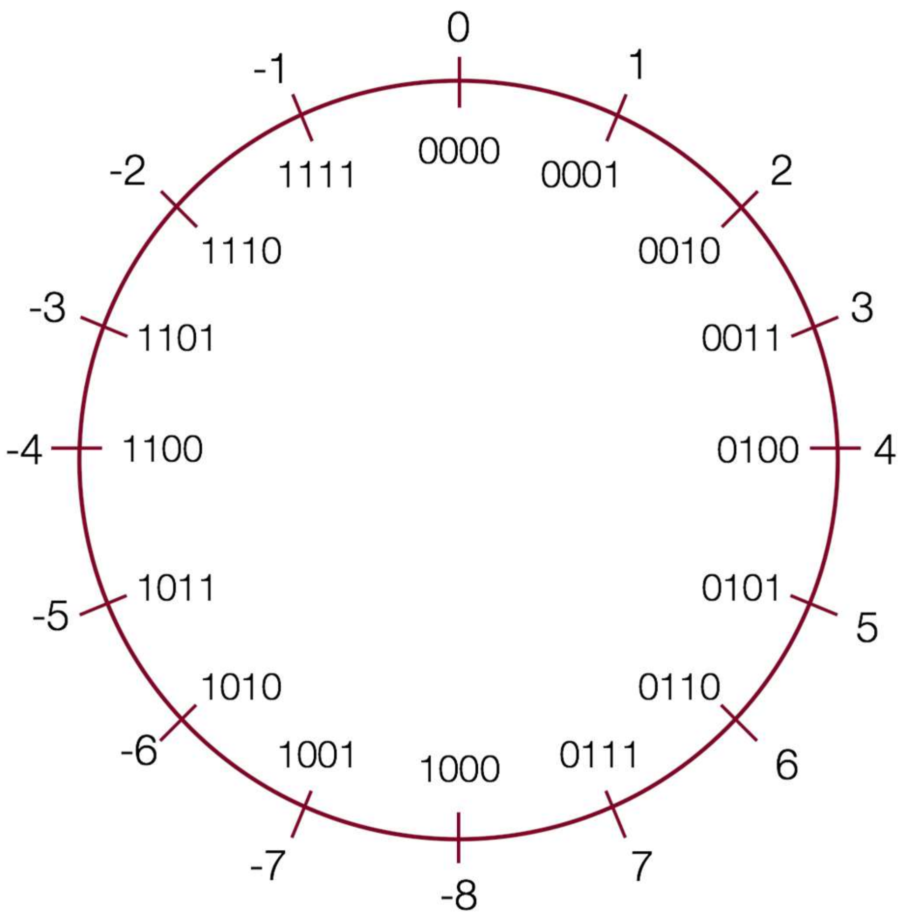
Shortcut: start from the right, and write down numbers until you get to a 1:

1

Now invert all the rest of the digits:

0101

Two's Complement: Neat Properties

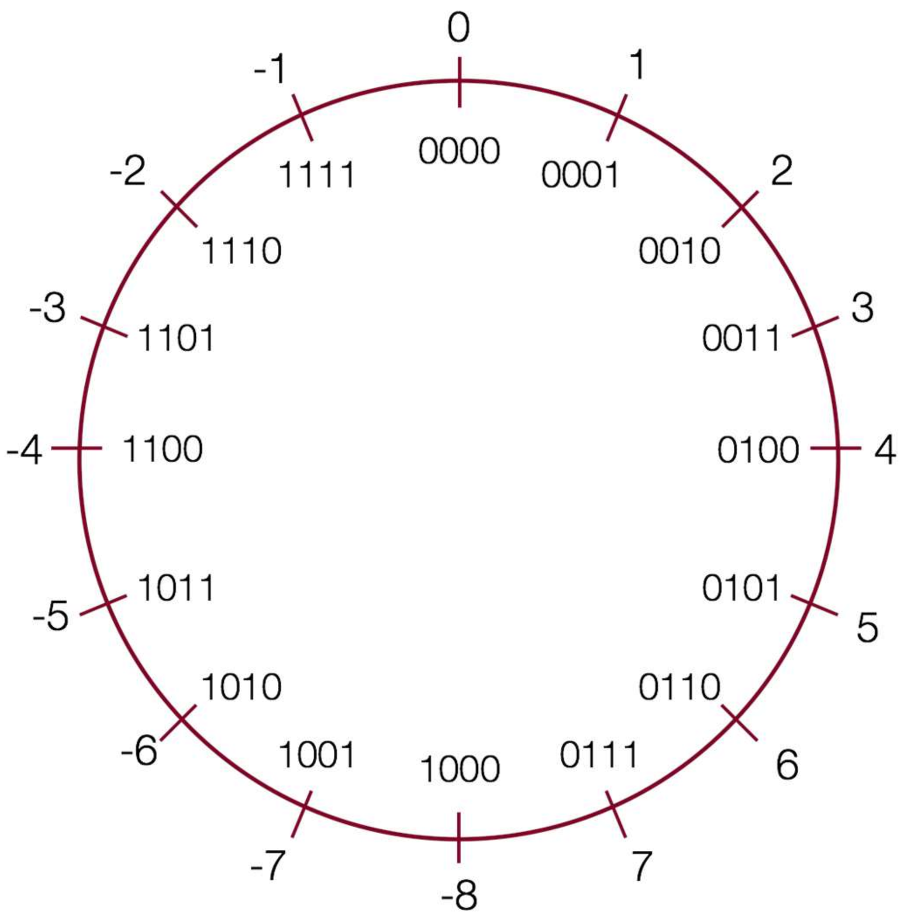


There are a number of useful properties associated with two's complement numbers:

1. There is only one zero (yay!)
2. The highest order bit (left-most) is 1 for negative, 0 for positive (so it is easy to tell if a number is negative)
3. Adding two numbers is just…adding! Example:
   2 + -5 = -3

  0010 ☞ 2
+1011 ☞ -5
  1101 ☞ -3 decimal (wow!)

Two's Complement: Neat Properties



More useful properties:

4. Subtracting two numbers is simply performing the two's complement on one of them and then adding. Example:
   4 - 5 = -1

0100 ☞ 4, 0101 ☞ 5

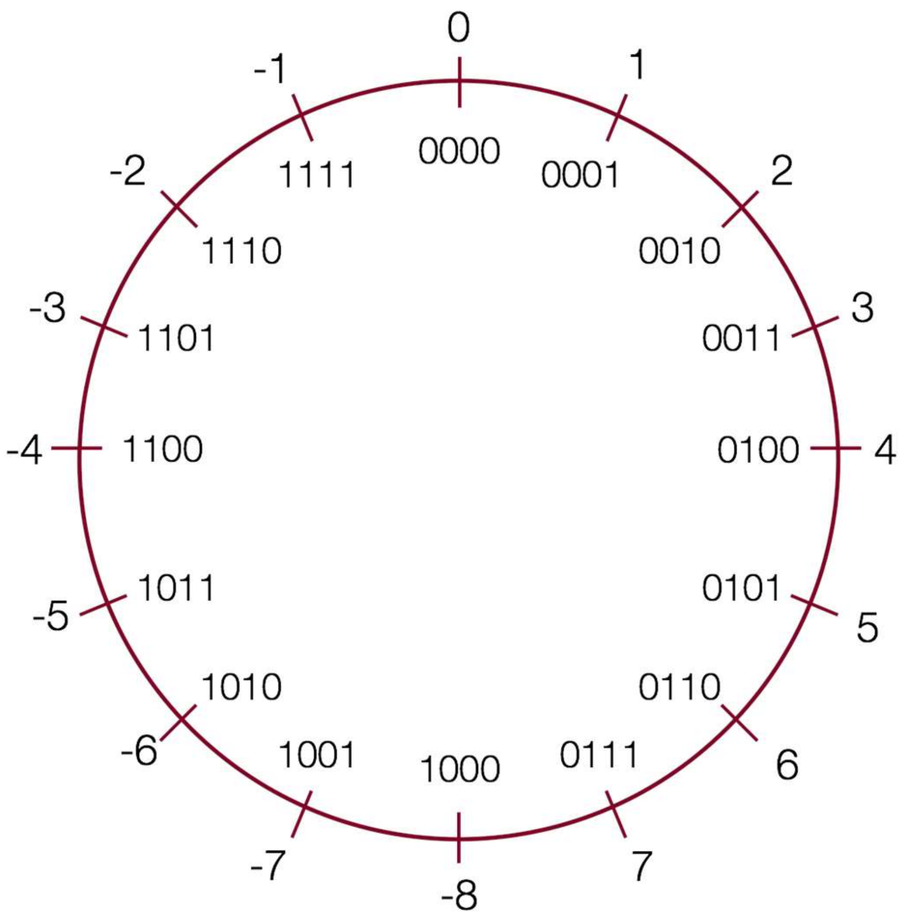
Find the two's complement of 5: 1011
add:

0100 ☞ 4

+1011 ☞ -5

1111 ☞ -1 decimal

Two's Complement: Neat Properties



More useful properties:

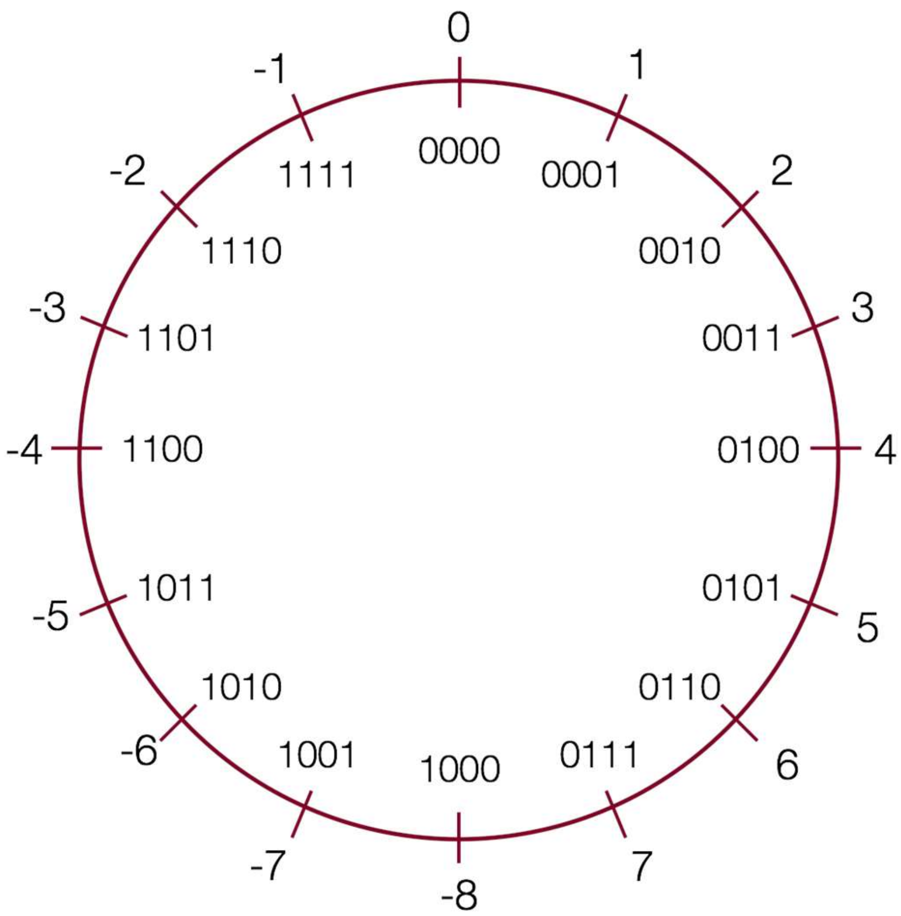
5. Multiplication of two's complement works just by multiplying (throw away overflow digits).

Example: -2 * -3 = 6

```
   1110 ☞ -2
  x1101 ☞ -3
   1110
  0000
 1110
+1110
10110110 ☞ 6
```

In the result line, ~~1011~~**0110** ☞ 6: the first four digits are struck through and the last four are bold.

Practice



Convert the following 4-bit numbers from positive to negative, or from negative to positive using two's complement notation:

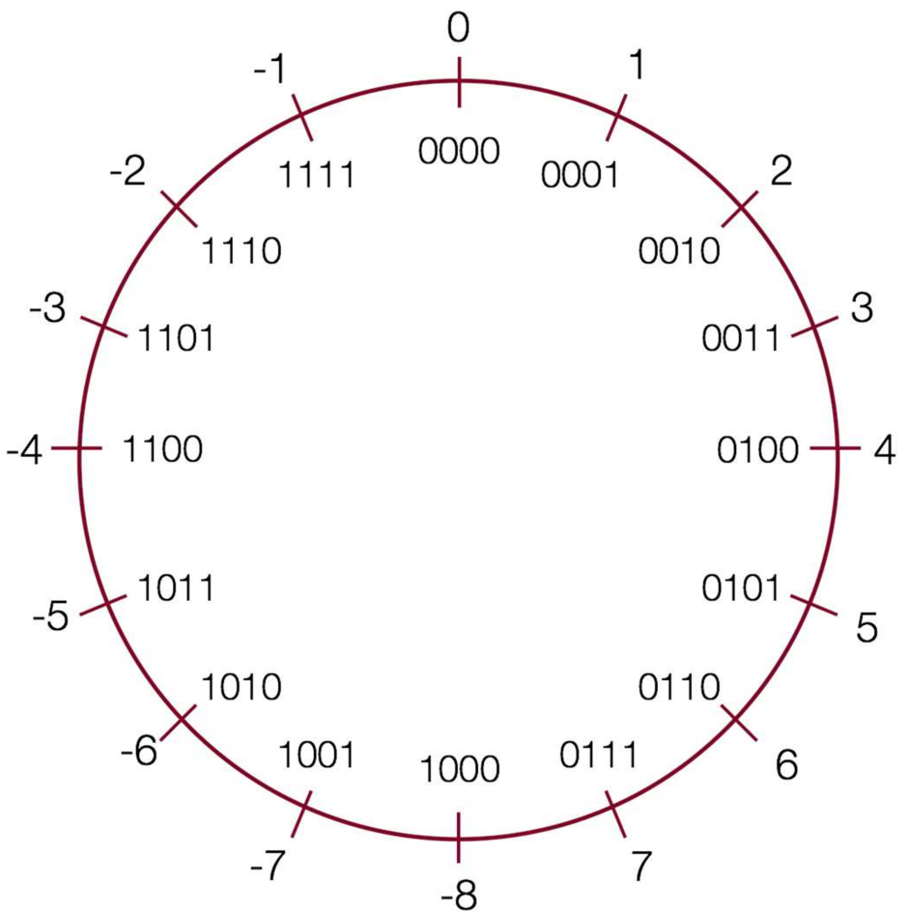
a. -4 (1100) ☞

b. 7 (0111) ☞

c. 3 (0011) ☞

d. -8 (1000) ☞

Practice



Convert the following 4-bit numbers from positive to negative, or from negative to positive using two's complement notation:

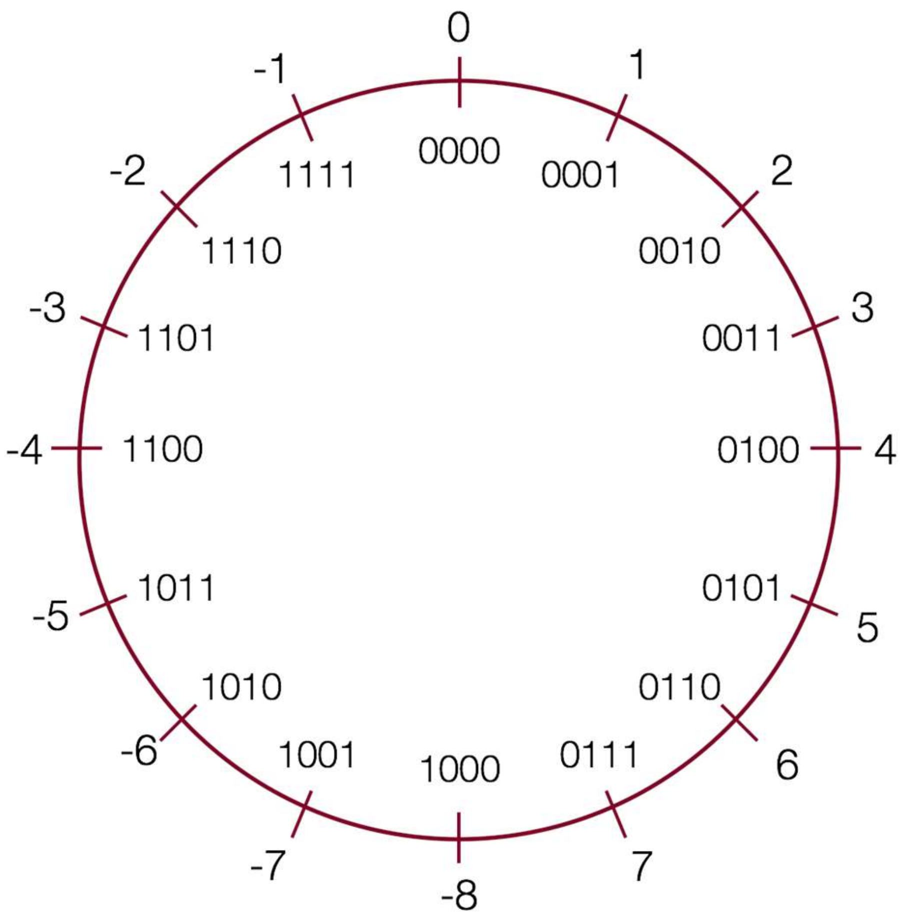
a. -4 (1100) ☞ 0100

b. 7 (0111) ☞ 1001

c. 3 (0011) ☞ 1101

d. -8 (1000) ☞ 1000 (?! If you look at the chart, +8 cannot be represented in two's complement with 4 bits!)

Practice



Convert the following 8-bit numbers from positive to negative, or from negative to positive using two's complement notation:

a. -4 (11111100) ☞ 00000100

b. 27 (00011011) ☞ 11100101

c. -127 (10000001) ☞ 01111111

d. 1 (00000001) ☞ 11111111

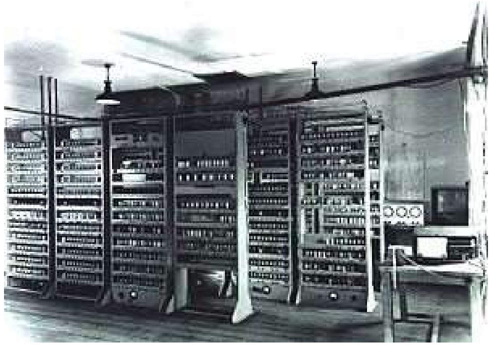
History: Two's complement

- The binary representation was first proposed by John von Neumann in *First Draft of a Report on the EDVAC* (1945)
  - That same year, he also invented the merge sort algorithm
- Many early computers used sign-magnitude or one's complement

+7	`0b0000 0111`
-7	`0b1111 1000`

8-bit one's complement

- The System/360, developed by IBM in 1964, was widely popular (had 1024KB memory) and established two's complement as the dominant binary representation of integers



EDSAC (1949)

System/360 (1964)

Casting Between Signed and Unsigned

Converting between two numbers in C can happen explicitly (using a parenthesized cast), or implicitly (without a cast):

explicit

```
int tx, ty;
unsigned ux, uy;
…
tx = (int) ux;
uy = (unsigned) ty;
```

implicit

```
int tx, ty;
unsigned ux, uy;
…
tx = ux; // cast to signed
uy = ty; // cast to unsigned
```

When casting: **the underlying bits do not change**, so there isn't any conversion going on, except that the variable is treated as the type that it is.

NOTE: Converting a signed number to unsigned preserves the bits not the number!

Casting Between Signed and Unsigned

When casting: **the underlying bits do not change**, so there isn't any conversion going on, except that the variable is treated as the type that it is. You cannot convert a signed number to its unsigned counterpart using a cast!

```
// test_cast.c
#include<stdio.h>
#include<stdlib.h>

int main() {
    int v = -12345;
    unsigned int uv = (unsigned int) v;

    printf("v = %d, uv = %u\n",v,uv);

    return 0;
}
```

```
$ ./test_cast
v = -12345, uv = 4294954951
```

Signed -> Unsigned
-12345 goes to 4294954951

Not 12345

IMPORTANT NOTE

- Because Types are just about how we read memory, it is important to note that casting does not impact the values or bits only the meaning that we expect them to have

- BEWARE: Expectations are like assumptions they can be violated or incorrect

Casting Between Signed and Unsigned

`printf` has three 32-bit integer representations:

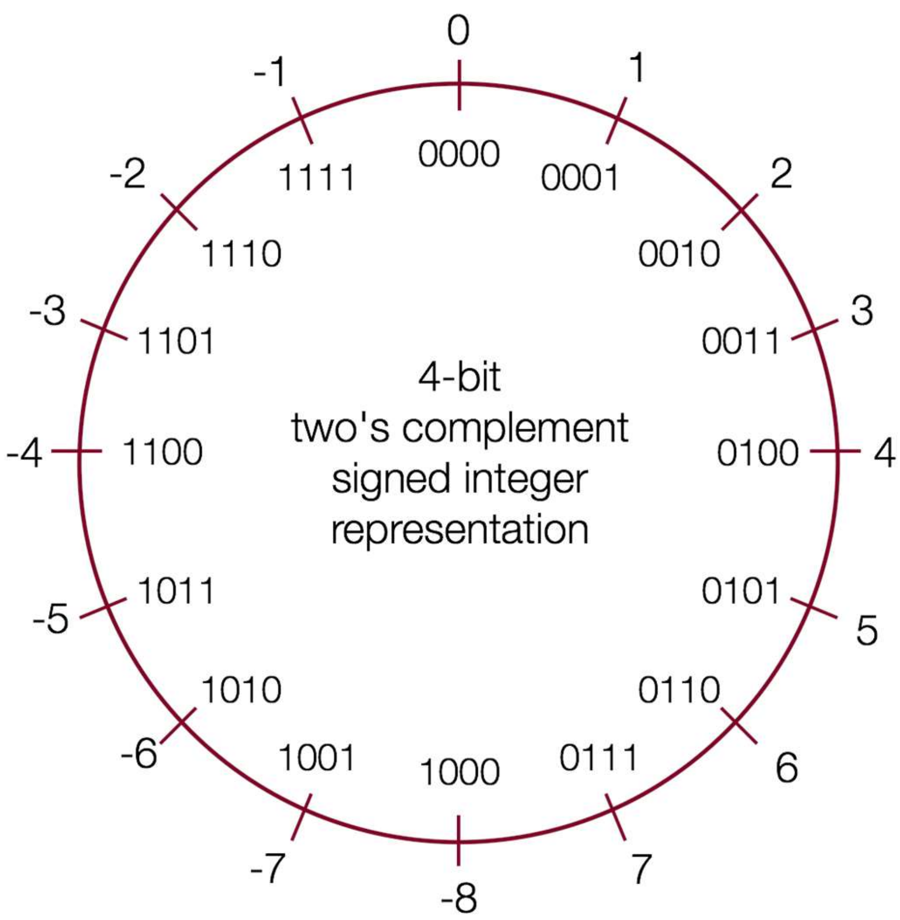
`%d` : signed 32-bit int
`%u` : unsigned 32-bit int
`%x` : hex 32-bit int

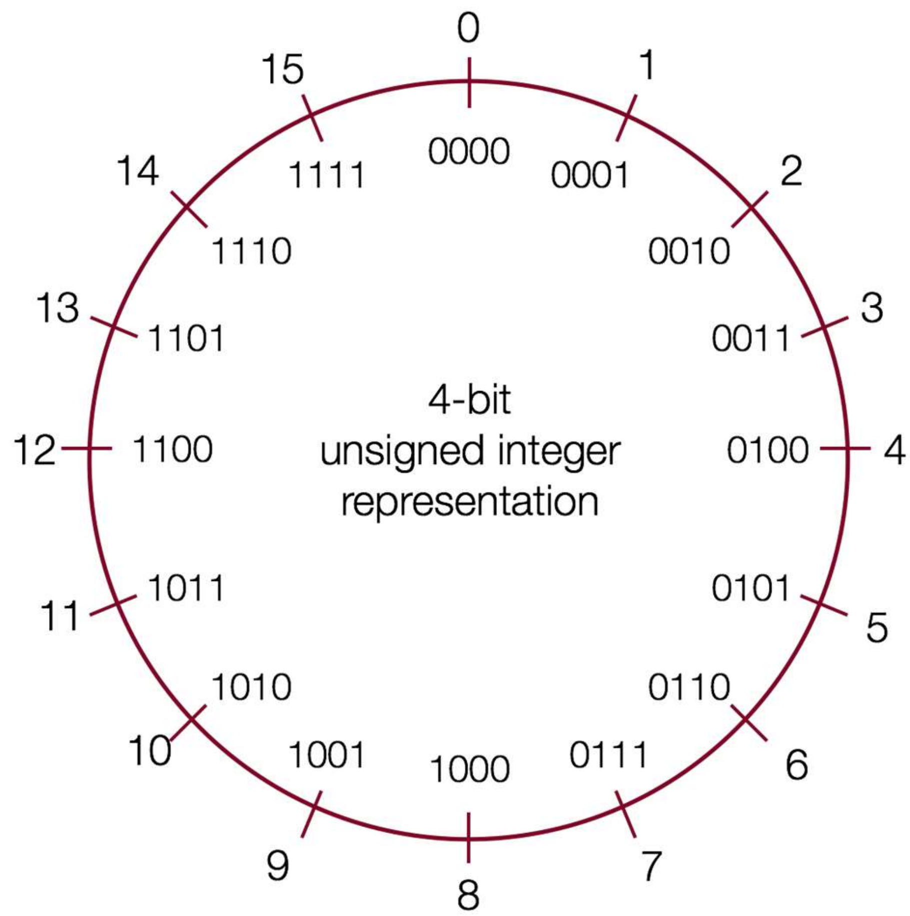
As long as the value is a 32-bit type, printf will treat it according to the formatter it is applying:

```
    int x = -1;
    unsigned u = 3000000000; // 3 billion

    printf("x = %u = %d\n", x, x);
    printf("u = %u = %d\n", u, u);
```

```
$ ./test_printf
x = 4294967295 = -1
u = 3000000000 = -1294967296
```

Signed vs Unsigned Number Wheels



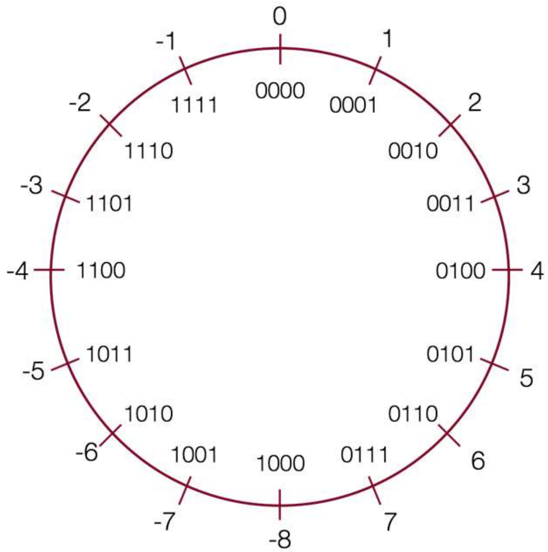


Comparison between signed and unsigned integers

When a C expression has combinations of signed and unsigned variables, you need to be careful!

If an operation is performed that has both a signed and an unsigned value, **C implicitly casts the signed argument to unsigned** and performs the operation assuming both numbers are non-negative. Let's take a look…

Expression	Type	Evaluation
`0 == 0U`		
`-1 < 0`		
`-1 < 0U`		
`2147483647 > -2147483647 - 1`		
`2147483647U > -2147483647 - 1`		
`2147483647 > (int)2147483648U`		
`-1 > -2`		
`(unsigned)-1 > -2`		

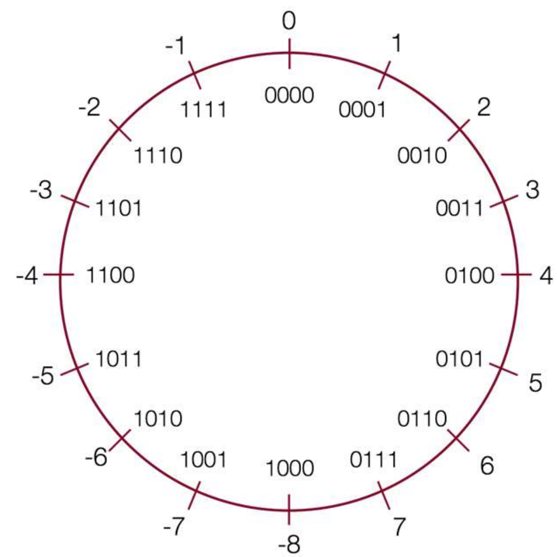


Comparison between signed and unsigned integers

When a C expression has combinations of signed and unsigned variables, you need to be careful!

If an operation is performed that has both a signed and an unsigned value, **C implicitly casts the signed argument to unsigned** and performs the operation assuming both numbers are non-negative. Let's take a look…

`Expression`	Type	Evaluation
`0 == 0U`	Unsigned	1
`-1 < 0`	Signed	1
`-1 < 0U`	Unsigned	0
`2147483647 > -2147483647 - 1`	Signed	1
`2147483647U > -2147483647 - 1`	Unsigned	0
`2147483647 > (int)2147483648U`	Signed	1
`-1 > -2`	Signed	1
`(unsigned)-1 > -2`	Unsigned	1



Note: In C, 0 is false and everything else is true. When C produces a boolean value, it allways chooses 1 to represent true.

Comparison between signed and unsigned integers

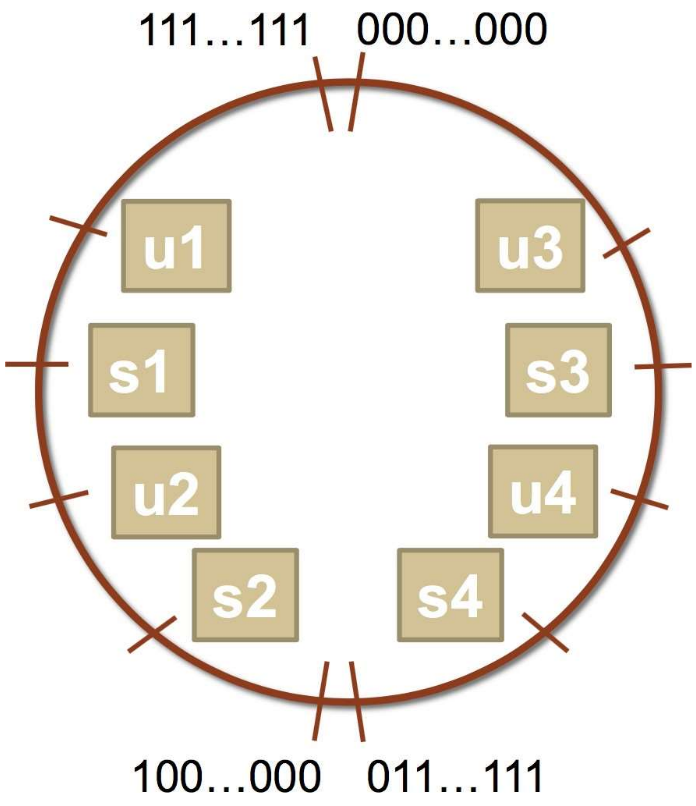
Let's try some more…a bit more abstractly.

```
int s1, s2, s3, s4;
unsigned int u1, u2, u3, u4;
```

What is the value of this expression?

```
u1 > s3
```

Go to https://pollev.com/akeppler



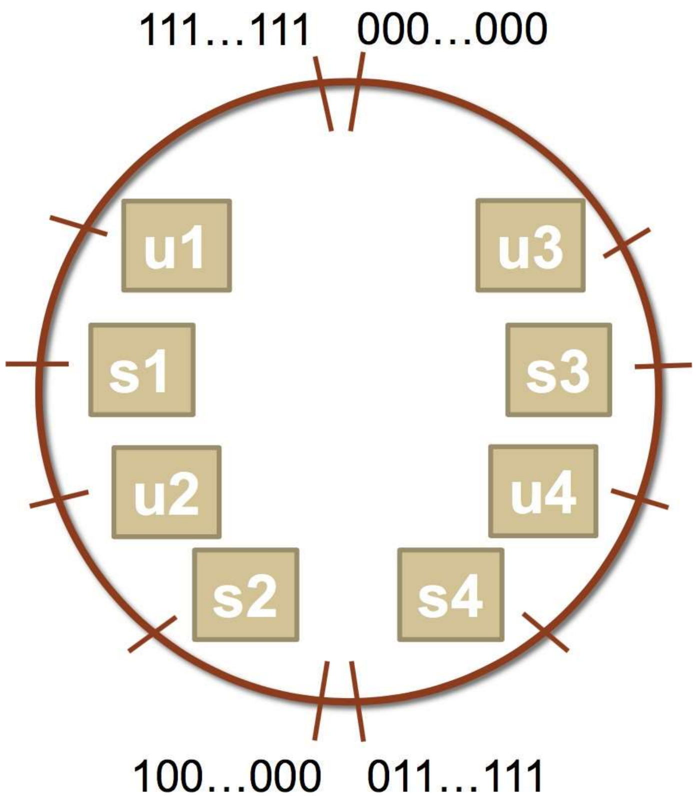
Comparison between signed

Let's try some more…a bit more abstractly.

```
int s1, s2, s3, s4;
unsigned int u1, u2, u3, u4;
```

Which many of the following statements are true? (*assume that variables are set to values that place them in the spots shown*)

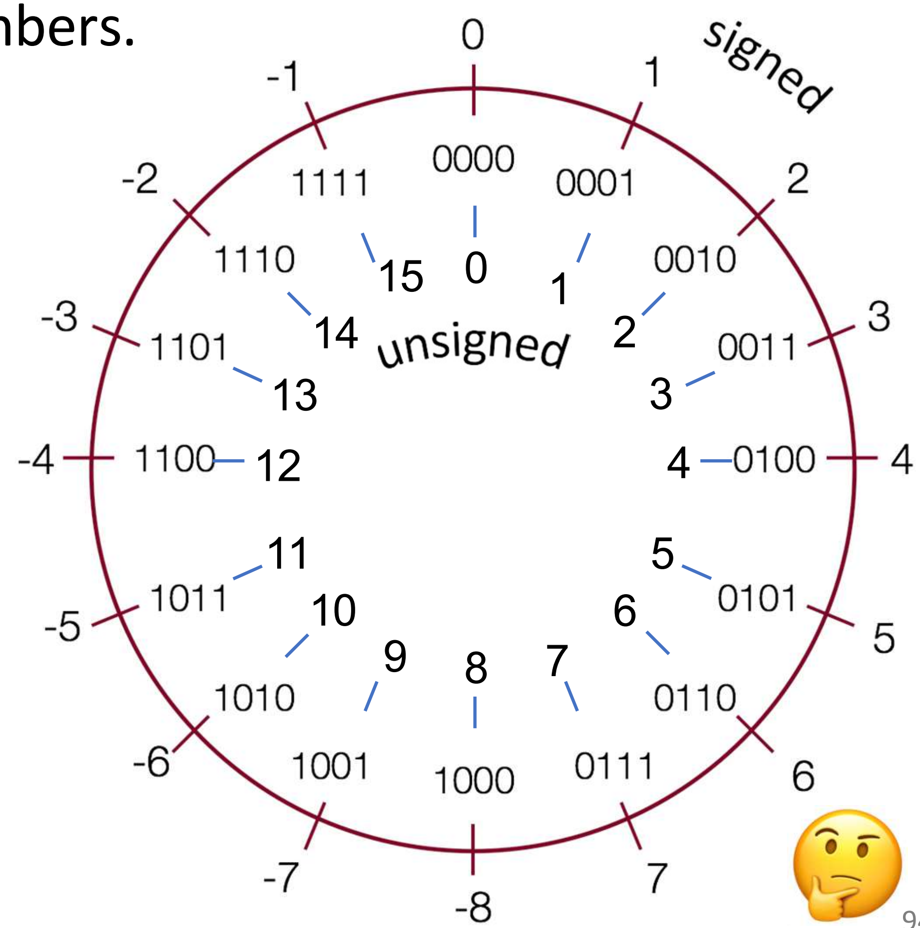
```
u1 > s3 : true
```



Overflow

- What is happening here? Assume 4-bit numbers.

```
  0b1101
+ 0b0100
________
```

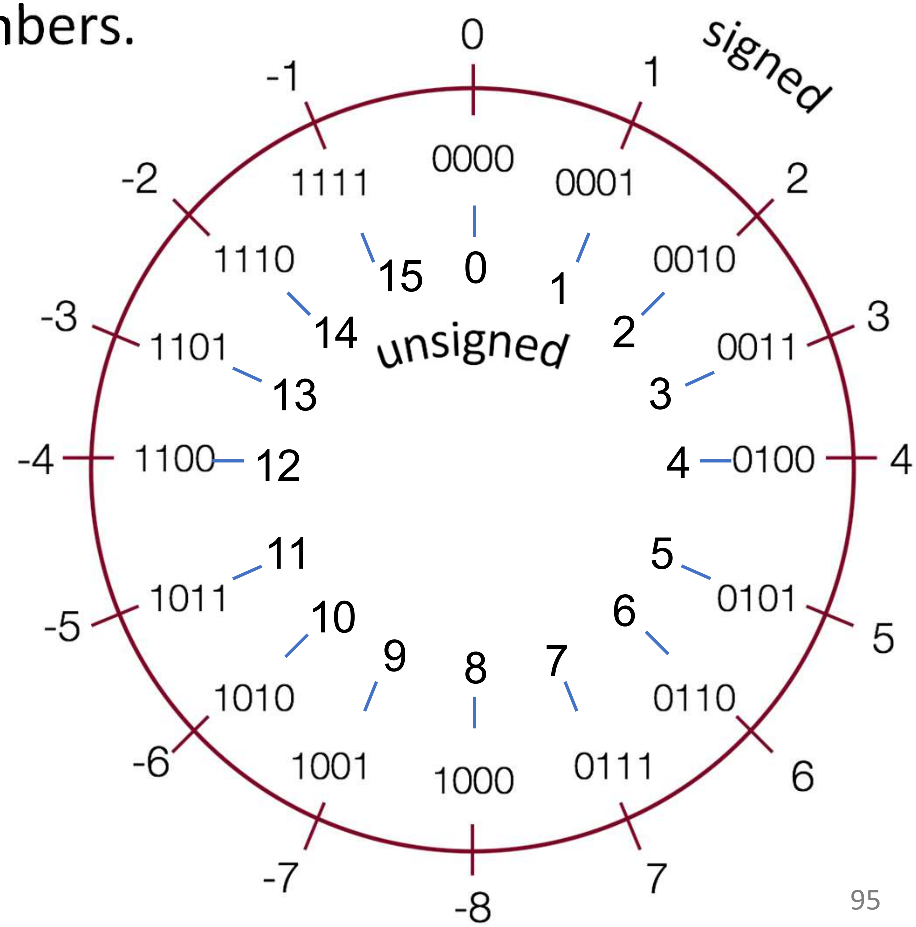


Overflow

- What is happening here? Assume 4-bit numbers.

```
  0b1101
+ 0b0100
```

Signed	Unsigned
`-3 + 4 = 1`	`13 + 4 = 1`
No overflow	Overflow



Limits and Comparisons

1. What is the…

	Largest unsigned?	Largest signed?	Smallest signed?
`char`			
`int`			

2. Will the following `char` comparisons evaluate to `true` or `false`?

   i. `-7 < 4`

   ii. `-7 < 4U`

   iii. `(char) 130 > 4`

   iv. `(char) -132 > 2`

Limits and Comparisons

1. What is the…

	Largest unsigned?	Largest signed?	Smallest signed?
`char`	$2^8 - 1 = 255$	$2^7 - 1 = 127$	$-2^7 = -128$
`int`	$2^{32} - 1 = 4294967296$	$2^{31} - 1 = 2147483647$	$-2^{31} = -2147483648$

These are available as UCHAR_MAX, INT_MIN, INT_MAX, etc. in the `<limits.h>` header.

Limits and Comparisons

2. Will the following `char` comparisons evaluate to `true` or `false`?

   i. `-7 < 4` **`true`**

   ii. `-7 < 4U` **`false`**

   iii. `(char) 130 > 4` **`false`**

   iv. `(char) -132 > 2` **`true`**

By default, numeric constants in C are signed ints, unless they are suffixed with u (unsigned) or L (long).

The `sizeof` Operator

```
long sizeof(type);

    // Example
    long int_size_bytes = sizeof(int);   // 4
    long short_size_bytes = sizeof(short); // 2
    long char_size_bytes = sizeof(char);   // 1
```

`sizeof` takes a variable type as a parameter and returns the size of that type, in bytes.

The `sizeof` Operator

As we have seen, integer types are limited by the number of bits they hold. On the 64-bit myth machines, we can use the `sizeof` operator to find how many bytes each type uses:

```
int main() {
    printf("sizeof(char): %d\n", (int) sizeof(char));
    printf("sizeof(short): %d\n", (int) sizeof(short));
    printf("sizeof(int): %d\n", (int) sizeof(int));
    printf("sizeof(unsigned int): %d\n", (int) sizeof(unsigned int));
    printf("sizeof(long): %d\n", (int) sizeof(long));
    printf("sizeof(long long): %d\n", (int) sizeof(long long));
    printf("sizeof(size_t): %d\n", (int) sizeof(size_t));
    printf("sizeof(void *): %d\n", (int) sizeof(void *));
    return 0;
}
```

```
$ ./sizeof
sizeof(char): 1
sizeof(short): 2
sizeof(int): 4
sizeof(unsigned int): 4
sizeof(long): 8
sizeof(long long): 8
sizeof(size_t): 8
sizeof(void *): 8
```

Type	**Width in bytes**	**Width in bits**
char	1	8
short	2	16
int	4	32
long	8	64
void *	8	64

MIN and MAX values for integers

Because we now know how bit patterns for integers works, we can figure out the maximum and minimum values, designated by `INT_MAX`, `UINT_MAX`, `INT_MIN`, (etc.), which are defined in `limits.h`

Type	**Width (bytes)**	**Width (bits)**	**Min in hex (name)**	**Max in hex (name)**
char	1	8	80 (CHAR_MIN)	7F (CHAR_MAX)
unsigned char	1	8	0	FF (UCHAR_MAX)
short	2	16	8000 (SHRT_MIN)	7FFF (SHRT_MAX)
unsigned short	2	16	0	FFFF (USHRT_MAX)
int	4	32	80000000 (INT_MIN)	7FFFFFFF (INT_MAX)
unsigned int	4	32	0	FFFFFFFF (UINT_MAX)
long	8	64	8000000000000000 (LONG_MIN)	7FFFFFFFFFFFFFFF (LONG_MAX)
unsigned long	8	64	0	FFFFFFFFFFFFFFFF (ULONG_MAX)

Min and Max Integer Values

- You can also find constants in the standard library that define the max and min for each type on that machine(architecture)

- Visit <limits.h> or <cstdint.h> and look for variables like:

```
INT_MIN
INT_MAX
UINT_MAX
LONG_MIN
LONG_MAX
ULONG_MAX
…
```

Expanding Bit Representations

- Sometimes, we want to convert between two integers of different sizes (e.g. **`short`** to **`int`**, or **`int`** to **`long`**).
- We might not be able to convert from a bigger data type to a smaller data type, but we do want to always be able to convert from a **smaller** data type to a **bigger** data type.
- For **unsigned** values, we can add *leading zeros* to the representation ("zero extension")
- For **signed** values, we can *repeat the sign of the value* for new digits ("sign extension"
- Note: when doing <, >, <=, >= comparison between different size types, it will *promote to the larger type*.

Expanding the bit representation of a number

For signed values, we want the number to remain the same, just with more bits. In this case, we perform a "sign extension" by repeating the sign of the value for the new digits. E.g.,

```
short s = 4;
// short is a 16-bit format, so                       s = 0000 0000 0000 0100b

int i = s;

// conversion to 32-bit int, so i = 0000 0000 0000 0000 0000 0000 0000 0100b
```

— or —

```
short s = -4;
// short is a 16-bit format, so                       s = 1111 1111 1111 1100b

int i = s;
// conversion to 32-bit int, so i = 1111 1111 1111 1111 1111 1111 1111 1100b
```

Converting from a smaller type to a larger type is also often called promotion
I.E. the number was promoted from short to int

Sign-extension Example

```
// show_bytes() defined on pg. 45, Bryant and O'Halloran
int main() {
    short sx = -12345;         // -12345
    unsigned short usx = sx;   // 53191
    int x = sx;                // -12345
    unsigned ux = usx;         // 53191

    printf("sx = %d:\t", sx);
    show_bytes((byte_pointer) &sx, sizeof(short));
    printf("usx = %u:\t", usx);
    show_bytes((byte_pointer) &usx, sizeof(unsigned short));
    printf("x  = %d:\t", x);
    show_bytes((byte_pointer) &x, sizeof(int));
    printf("ux = %u:\t", ux);
    show_bytes((byte_pointer) &ux, sizeof(unsigned));

    return 0;
}
```

```
$ ./sign_extension
sx = -12345:   c7 cf
usx = 53191:   c7 cf
x  = -12345:   c7 cf ff ff
ux = 53191:    c7 cf 00 00
```

(careful: this was printed on the little-endian myth machines!)

Truncating Numbers: Signed

What if we want to reduce the number of bits that a number holds? E.g.

```
int x = 53191;          // 53191
short sx = (short) x; // -12345
int y = sx;
```

This is a form of *overflow*! We have altered the value of the number. Be careful!

We don't have enough bits to store the int in the short for the value we have in the `int`, so the strange values occur.

What is y above? We are converting a short to an int, so we sign-extend, and we get -12345!

`1100 1111 1100 0111` becomes

`1111 1111 1111 1111 1100 1111 1100 0111`

Play around here: http://www.convertforfree.com/twos-complement-calculator/

Truncating Numbers: Signed

If the number does fit into the smaller representation in the current form, it will convert just fine.

```
int x = -3;          // -3
short sx = (short) -3; // -3
int y = sx; // -3
```

x: `1111 1111 1111 1111 1111 1111 1111 1101` becomes
sx: `1111 1111 1111 1101`

Play around here: http://www.convertforfree.com/twos-complement-calculator/

Truncating Numbers: Unsigned

We can also lose information with unsigned numbers:

```
unsigned int x = 128000;
unsigned short sx = (short) x;
unsigned int y = sx;
```

Bit representation for x = 128000 (32-bit unsigned int):

```
0000 0000 0000 0001 1111 0100 0000 0000
```

Truncated unsigned short sx:

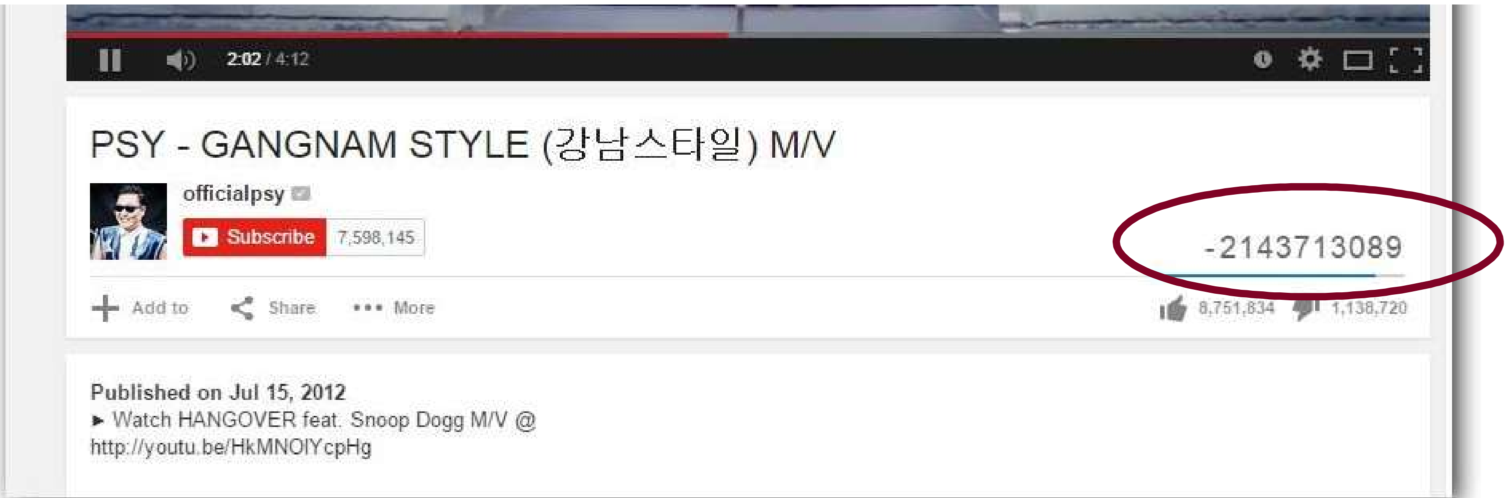
```
                    1111 0100 0000 0000
```

which equals 62464 decimal.

Converting back to an unsigned int, y = 62464

Overflow in Signed Addition

Signed overflow wraps around to the negative numbers:



YouTube fell into this trap — their view counter was a signed, 32-bit int. They fixed it after it was noticed, but for a while, the view count for Gangnam Style (the first video with over `INT_MAX` number of views) was negative.

Overflow In Practice: PSY

YouTube: “We never thought a video would be watched in numbers greater than a 32-bit integer (=2,147,483,647 views), but that was before we met PSY. "Gangnam Style" has been viewed so many times we had to upgrade to a 64-bit integer (9,223,372,036,854,775,808)!”

Overflow in Signed Addition

In the news on January 5, 2022 (!):

ars TECHNICA BIZ & IT TECH SCIENCE POLICY CARS GAMING & CUL

GOOD THING ANDROID IS GREAT AT ROLLING OUT UPDATES —

Google fixes nightmare Android bug that stopped user from calling 911

An integer overflow/underflow crash lets misbehaving apps lock users out of 911.

RON AMADEO - 1/5/2022, 3:09 PM

https://arstechnica.com/gadgets/2022/01/google-fixes-nightmare-android-bug-that-stopped-user-from-calling-911/

Overflow in Signed Addition

Signed overflow wraps around to the negative numbers.

```
#include<stdio.h>
#include<stdlib.h>
#include<limits.h> // for INT_MAX

int main() {
    int a = INT_MAX;
    int b = 1;
    int c = a + b;

    printf("a = %d\n",a);
    printf("b = %d\n",b);
    printf("a + b = %d\n",c);

    return 0;
}
```

```
$ ./signed_overflow
a = 2147483647
b = 1
a + b = -2147483648
```

***Technically**, signed integers in C **produce undefined behavior** when they overflow. On two's complement machines (virtually all machines these days), it does overflow predictably. You can test to see if your addition will be correct:*

```
// for addition
#include <limits.h>
int a = <something>;
int x = <something>;
if ((x > 0) && (a > INT_MAX - x)) /* `a + x` would overflow */;
if ((x < 0) && (a < INT_MIN - x)) /* `a + x` would underflow */;
```

Overflow

At which points can overflow occur for signed and unsigned int? *(assume binary values shown are all 32 bits)*

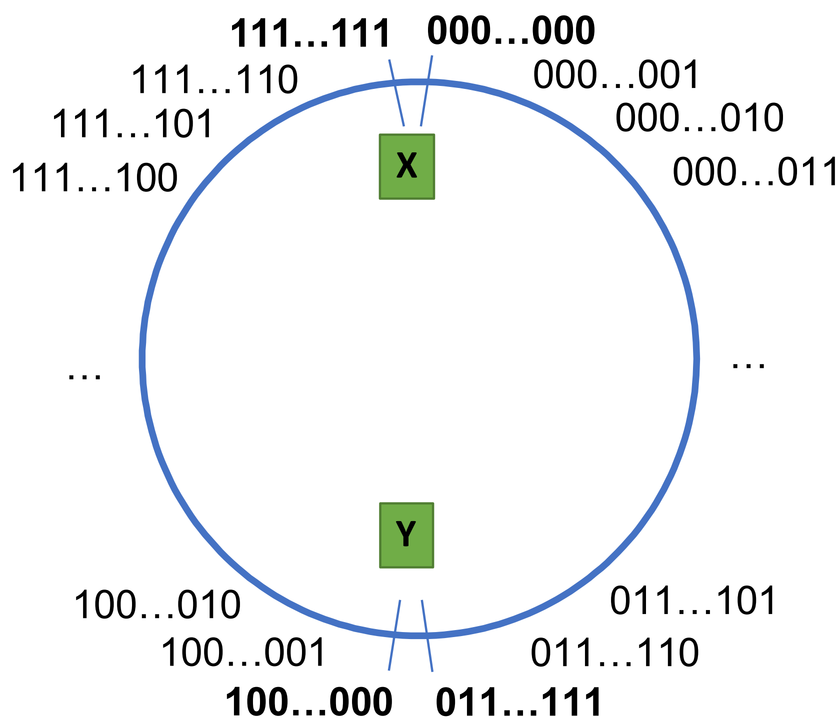
A. Signed and unsigned can both overflow at points X and Y

B. Signed can overflow only at X, unsigned only at Y

C. Signed can overflow only at Y, unsigned only at X

D. Signed can overflow at X and Y, unsigned only at X

E. Other

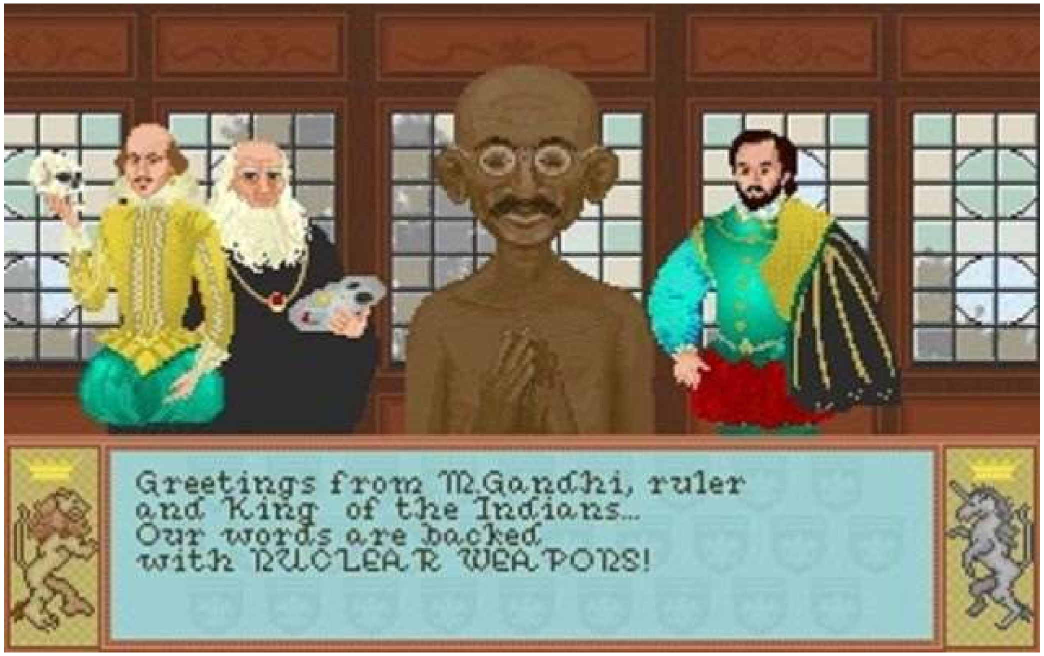


Overflow In Practice: Timestamps

- Many systems store timestamps as **the number of seconds since Jan. 1, 1970** in a **signed 32-bit integer**.
- **Problem:** the latest timestamp that can be represented this way is 3:14:07 UTC on Jan. 13 2038!

Overflow In Practice: Gandhi

- In the game "Civilization", each civilization leader had an "aggression" rating. Gandhi was meant to be peaceful, and had a score of 1.
- If you adopted "democracy", all players' aggression reduced by 2. Gandhi's went from 1 to **255**!
- Gandhi then became a big fan of nuclear weapons.



https://kotaku.com/why-gandhi-is-such-an-asshole-in-civilization-1653818245

Overflow in Practice:

- Pacman Level 256
- Make sure to reboot Boeing Dreamliners every 248 days
- Comair/Delta airline had to cancel thousands of flights days before Christmas
- Reported vulnerability CVE-2019-3857 in libssh2 may allow a hacker to remotely execute code
- Donkey Kong Kill Screen

3 Minute Break