Computer Systems

CS107

Cynthia Lee

Today's Topics

LAST TIME:

- Number representation
  - Integer representation
  - Signed numbers with two's complement

THIS TIME:

- Number representation
  - The integer number line for signed and unsigned
  - Overflow and underflow
  - Comparison, extension and truncation in signed and unsigned
  - Bitwise operations and bit sets

COMING UP:

- Today is last day of topics that will be included on next week's midterm
  - Practice exams and topics list are up now

Signed integers with two's complement representation

Review from Friday

Self-test review from Friday: Two's complement

What is the base-10 equivalent of the signed (two's complement) 4-bit binary number 1010?

a) -10
b) 10
c) 11
d) -11
e) 5
f) -5
g) 6
h) -6
i) Other

Signed integers with two's complement

Goal: write 5 in 8-bit two's complement

Steps to write a positive (or zero) number in two's complement:

1. Write the number in usual unsigned binary representation
2. Make sure that the number will "fit" in the number of bits you have
   - For positive numbers, there needs to be at least one zero in the most significant (leftmost) bit
   - 00000101 (no problem for 5 in 8 bits)
3. Done!
   - Answer: 00000101

Signed integers with two's complement

Goal: write -5 in 8-bit two's complement

Steps to write a negative number in two's complement:

1. Write the *absolute value* of the number in usual unsigned binary representation
2. Make sure that the number will "fit" in the number of bits you have
   - Since we are writing the absolute value, a positive number, there needs to be at least one zero in the most significant (leftmost) bit*
   - 00000101 (no problem for 5 in 8 bits)
3. "Flip" each bit (0 → 1, 1 → 0)
   - 00000101 → 11111010
4. Add one
   - 11111010 → 11111011
5. Done!
   - Answer: 11111011

* There is one negative number whose positive number won't "fit"—more on this Friday

Inbox (15) - cynthia.t | Stanford Center for F | integer - What is the

stackoverflow.com/questions/94591/what-is-the-maximum-value-for-a-int32

What is the maximum value for a int32?

606 · 92

I can never remember that number. I need a memory rule.

integer

share improve this question

edited May 28 '14 at 14:09
Ben Hoffstein
49.5k • 5 • 66 • 101

asked Sep 18 '08 at 17:18
Flinkman
5,181 • 4 • 18 • 48

107 Why would you need the exact number? I remember "(2^31)-1" or "+/- 2 billion" and that's good enough for everything I ever needed. – Joachim Sauer Mar 3 '09 at 11:21

27 unsigned: 2^{32}-1 = 4·1024^3-1; signed: -2^{31} .. +2^{31}-1, because the sign-bit is the highest bit. Just learn 2^0=1 to 2^{10}=1024 and combine. 1024=1k, 1024^2=1M, 1024^3=1G – comonad Mar 28 '11 at 20:01

6 I generally remember that every 3 bits is about a decimal digit. This gets me to the right order of magnitude: 32 bits is 10 digits. – Barmar Oct 2 '13 at 15:11

30 Answers

active oldest votes

2397

It's 2,147,483,647. Easiest way to memorize it is via a tattoo.

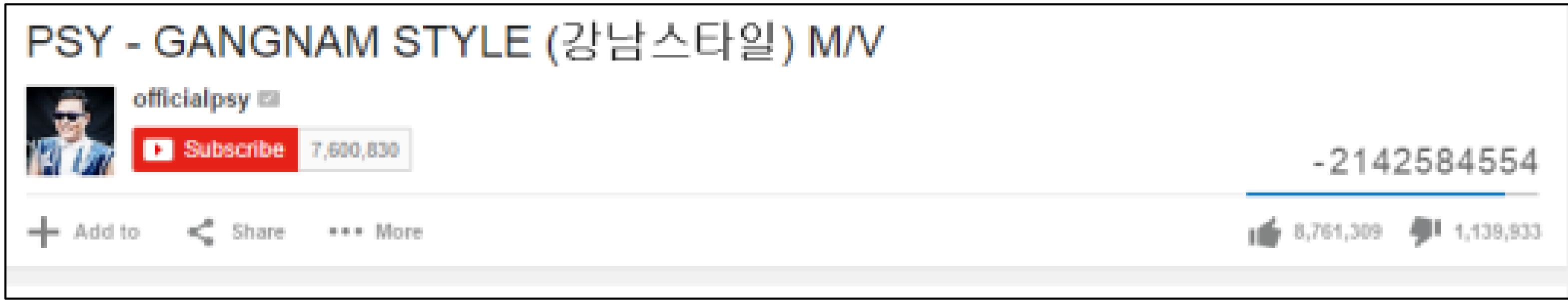
share improve this answer

edited Oct 20 '14 at 16:30
Allbite
1,415 • 1 • 13 • 15

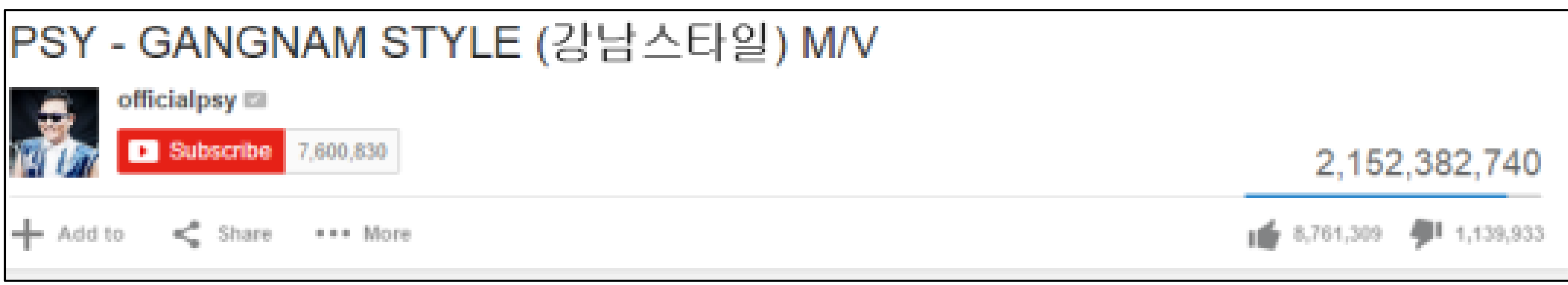
answered Sep 18 '08 at 17:20
Ben Hoffstein
49.5k • 5 • 66 • 101

Overflow in two's complement

- In two's complement, when you exceed the maximum value of int (2,147,483,647), you "wrap around" to negative numbers:



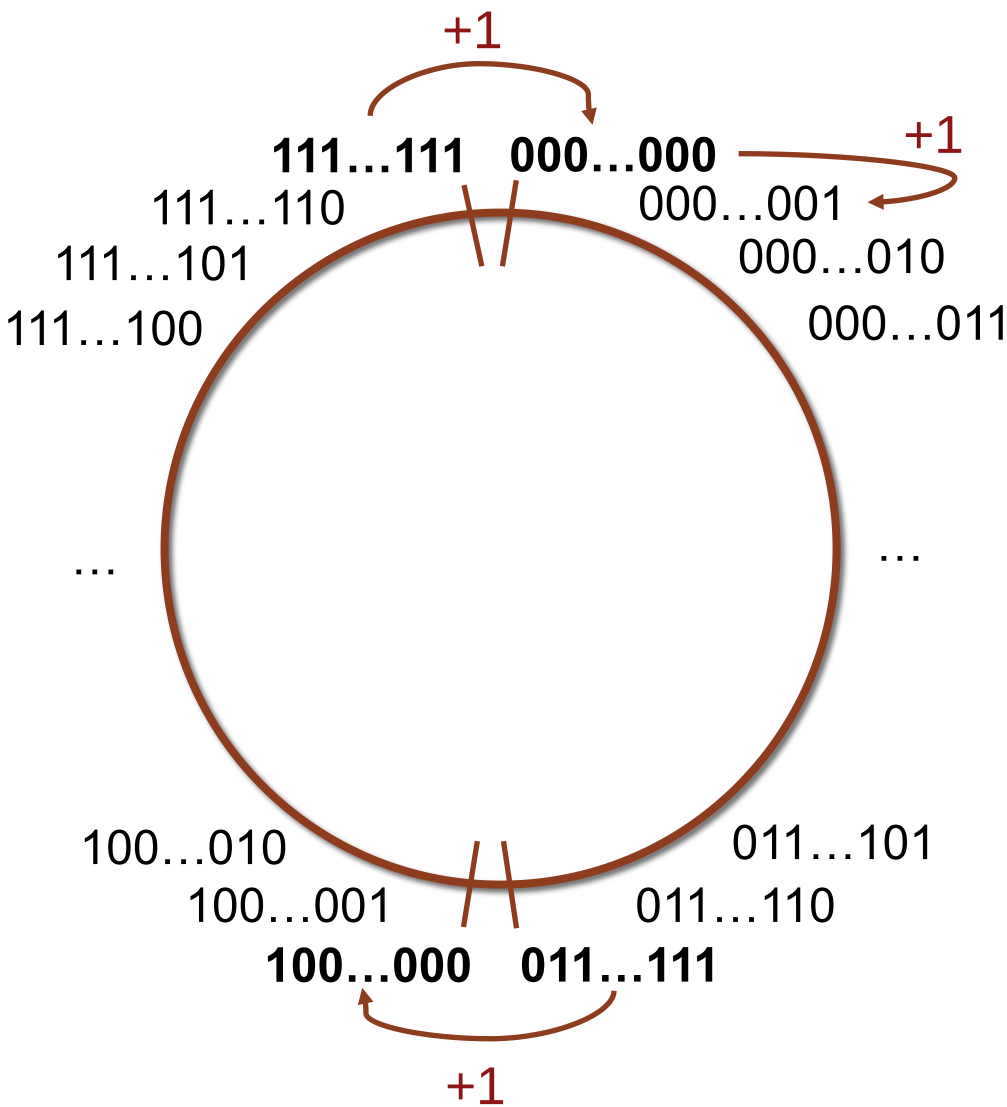
- Here is the link after Google upgraded to 64-bit integers:



Reasoning about signed and unsigned

Integer Representation

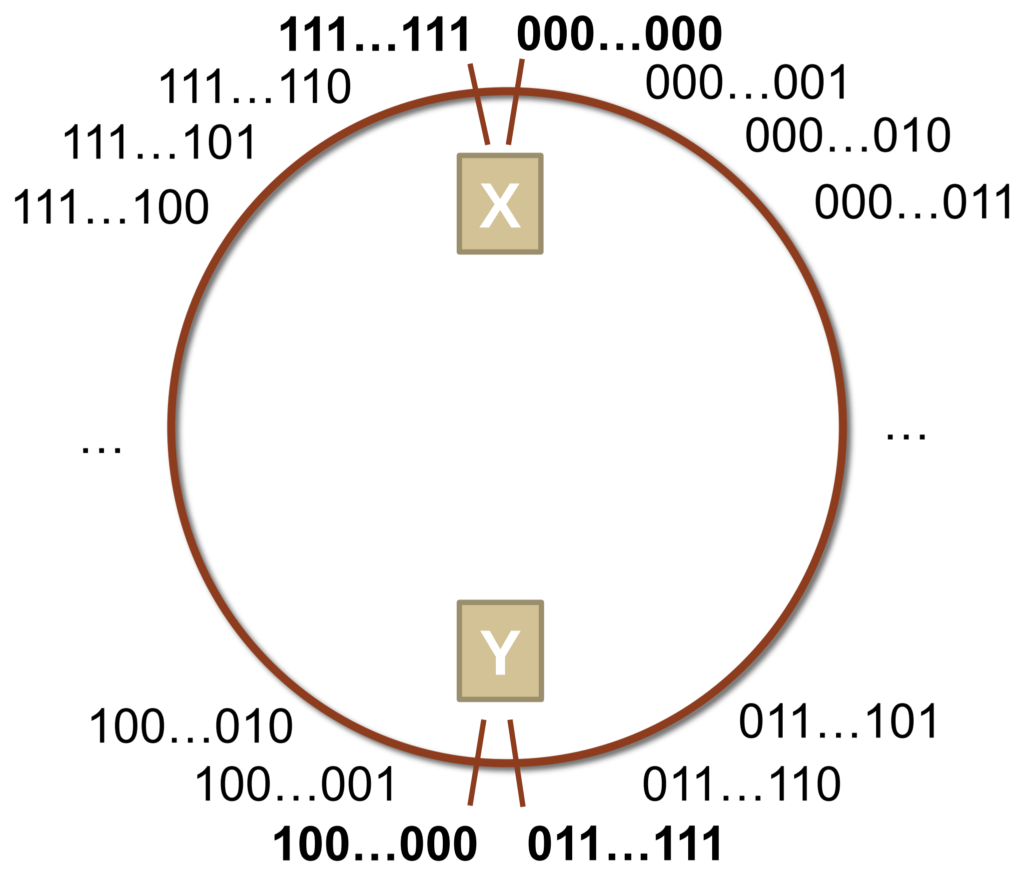
(assume binary values shown are all 32 bits)



Signed and unsigned numbers

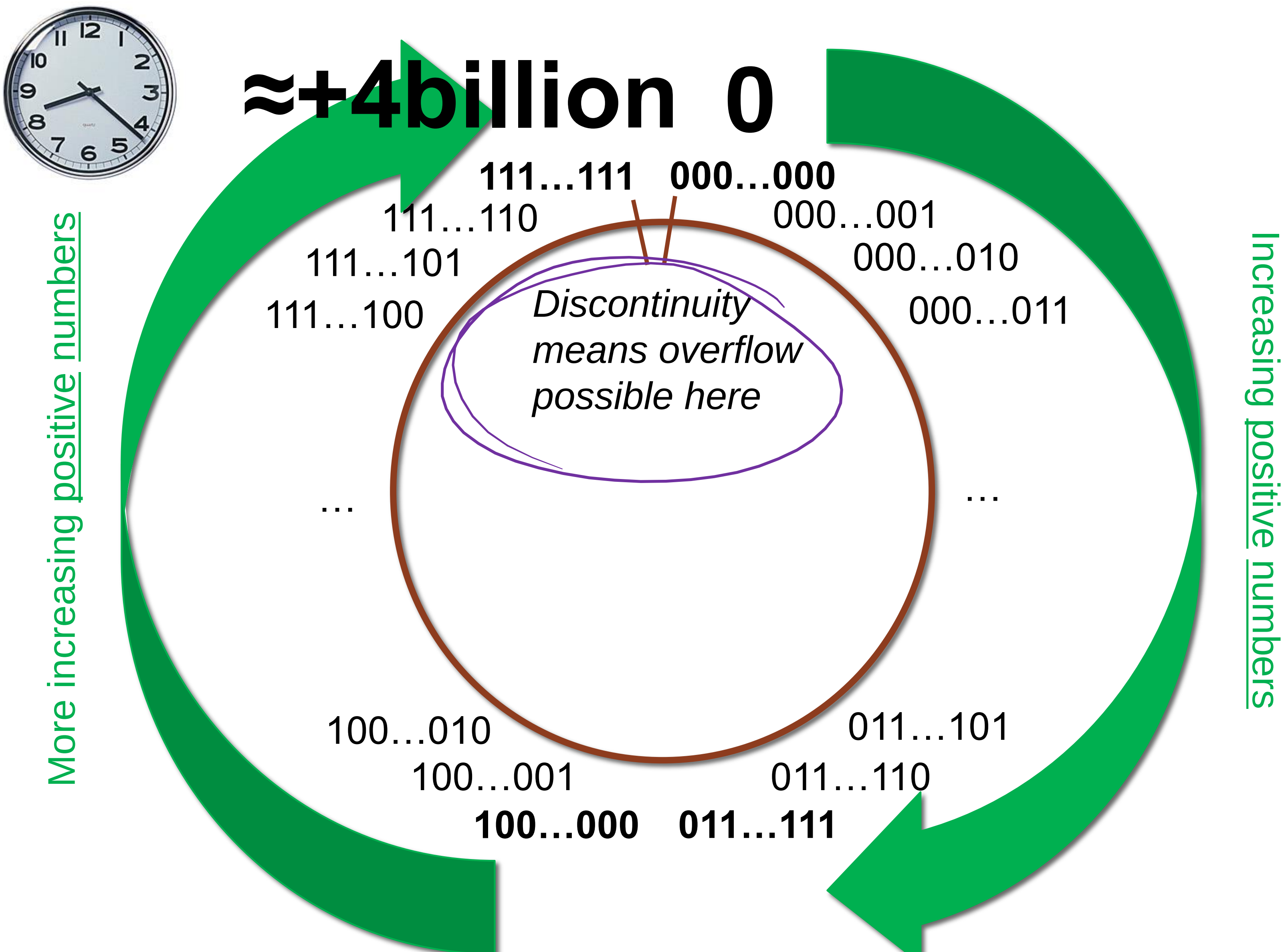
At which points can overflow occur for signed and unsigned int?
(assume binary values shown are all 32 bits)

A. Signed and unsigned can both overflow at points X and Y
B. Signed can overflow at X, unsigned at Y
C. Signed can overflow at Y, unsigned at X
D. Signed can overflow at X and Y, unsigned only at X
E. Other



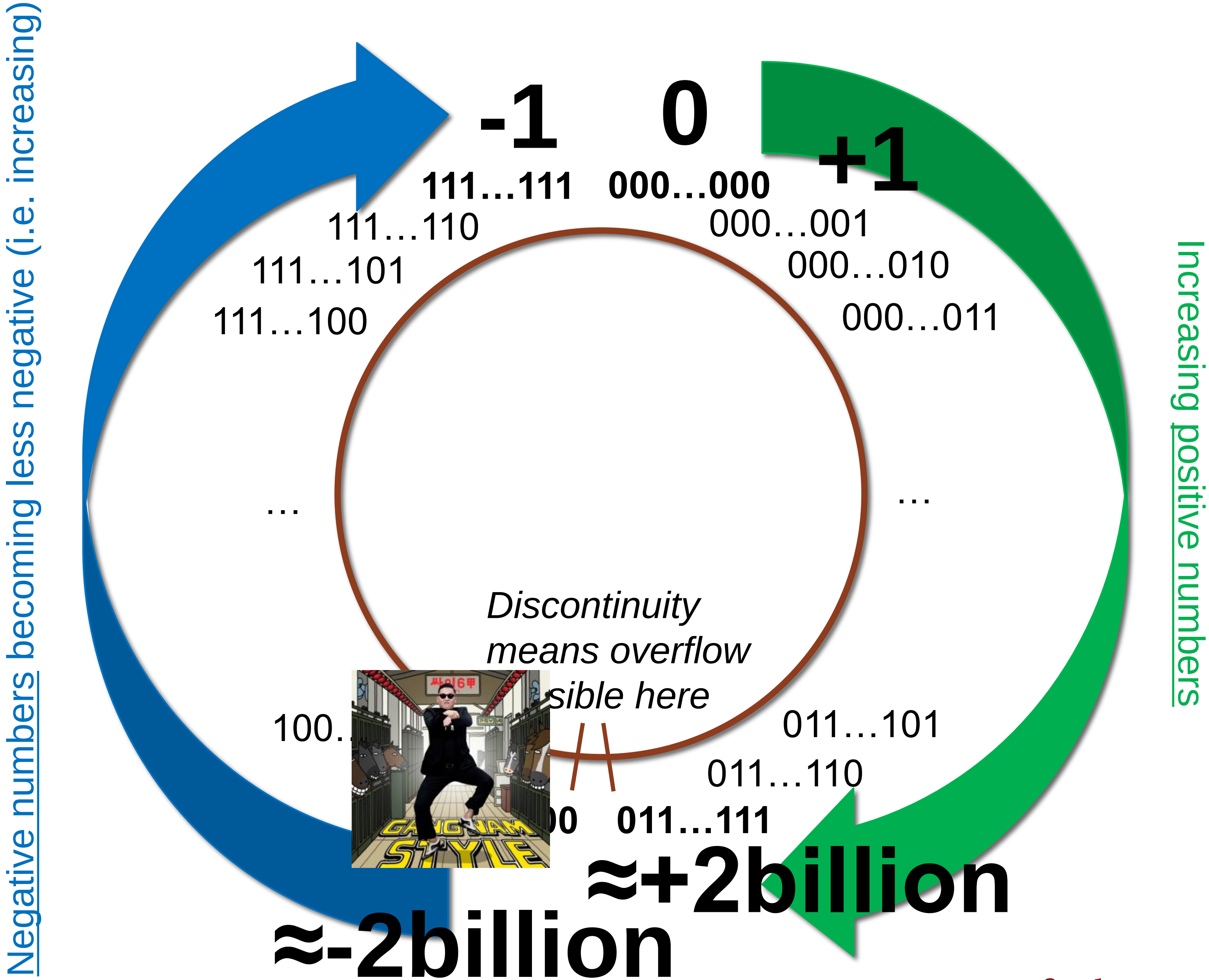
UNSIGNED integers

(assume binary values shown are all 32 bits)



SIGNED integers

(assume binary values shown are all 32 bits)



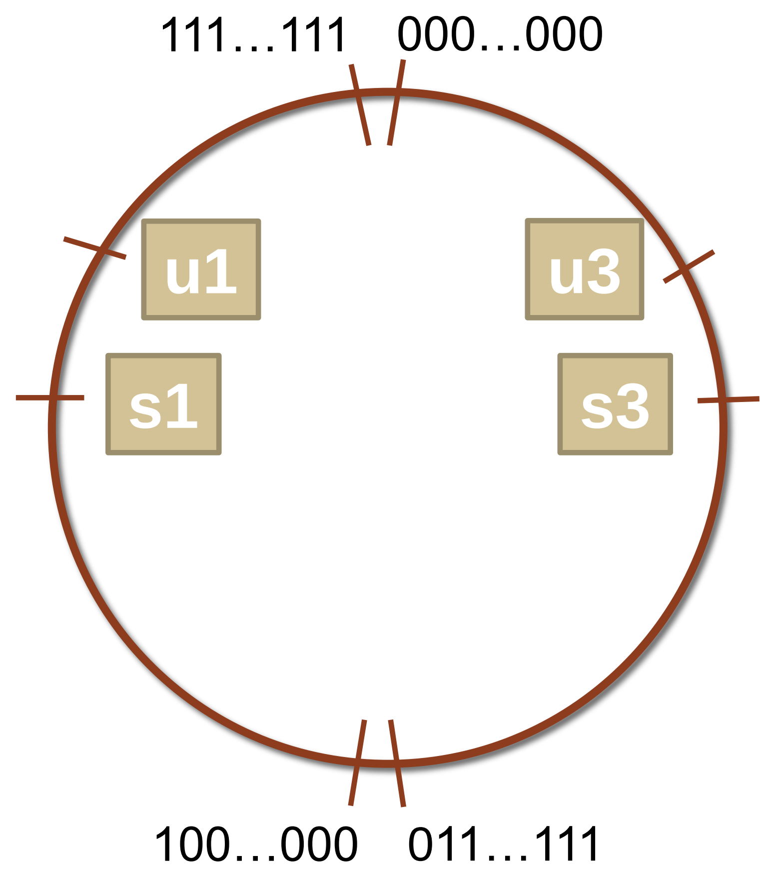
Comparison operators in signed and unsigned numbers

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

Are the following statements true?

(assume that variables are set to values that place them in the spots shown)

› `s3 > u3`	Easy: true
› `s1 > s3`	Easy: false
› `u1 > u3`	Easy: true
› `s1 > u3`	Hmmm!??!



C just needs to choose one or the other scheme to dominate. It chooses…drumroll…

unsigned!

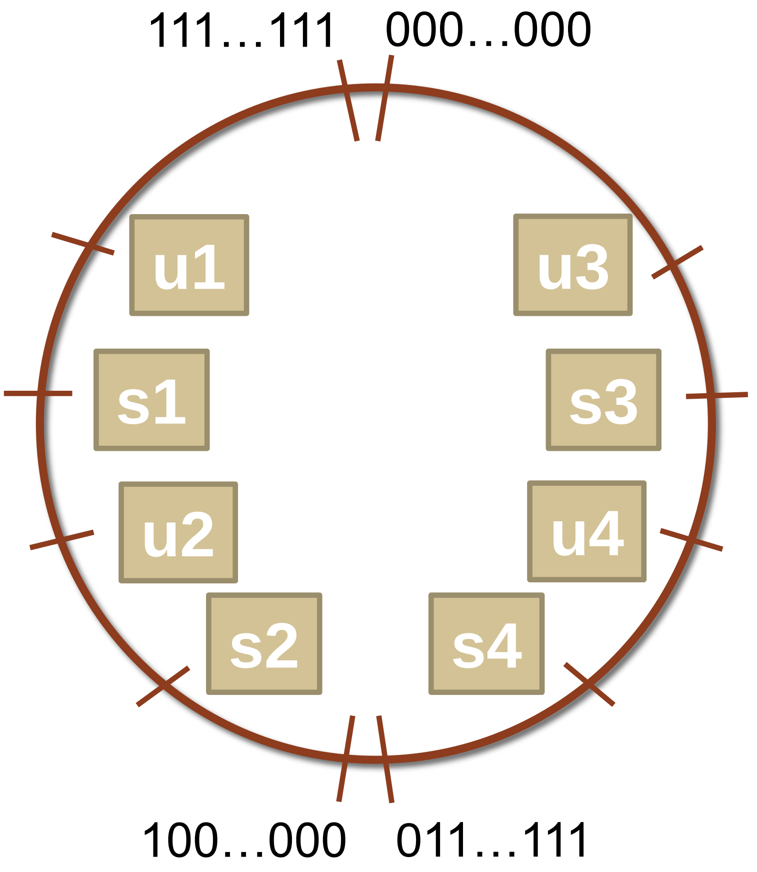
So this is **TRUE**.

HOME SELF-TEST:
Comparison operators in signed and unsigned numbers

```
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)*

- `s3 > u3`
- `u2 > u4`
- `s2 > s4`
- `s1 > s2`
- `u1 > u2`
- `s1 > u3`



Type truncation in the char/short/int/long family

```
int           i1 = 0x8000007F; // = -2147483521
int           i2 = 0x000000FF; // = 255
char          s1 = i1;         // = 0x7F = 127
char          s2 = i2;         // = 0xFF = -1
unsigned char u1 = i1;         // = 0x7F = 127
unsigned char u2 = i2;         // = 0xFF = 255
```

+ !

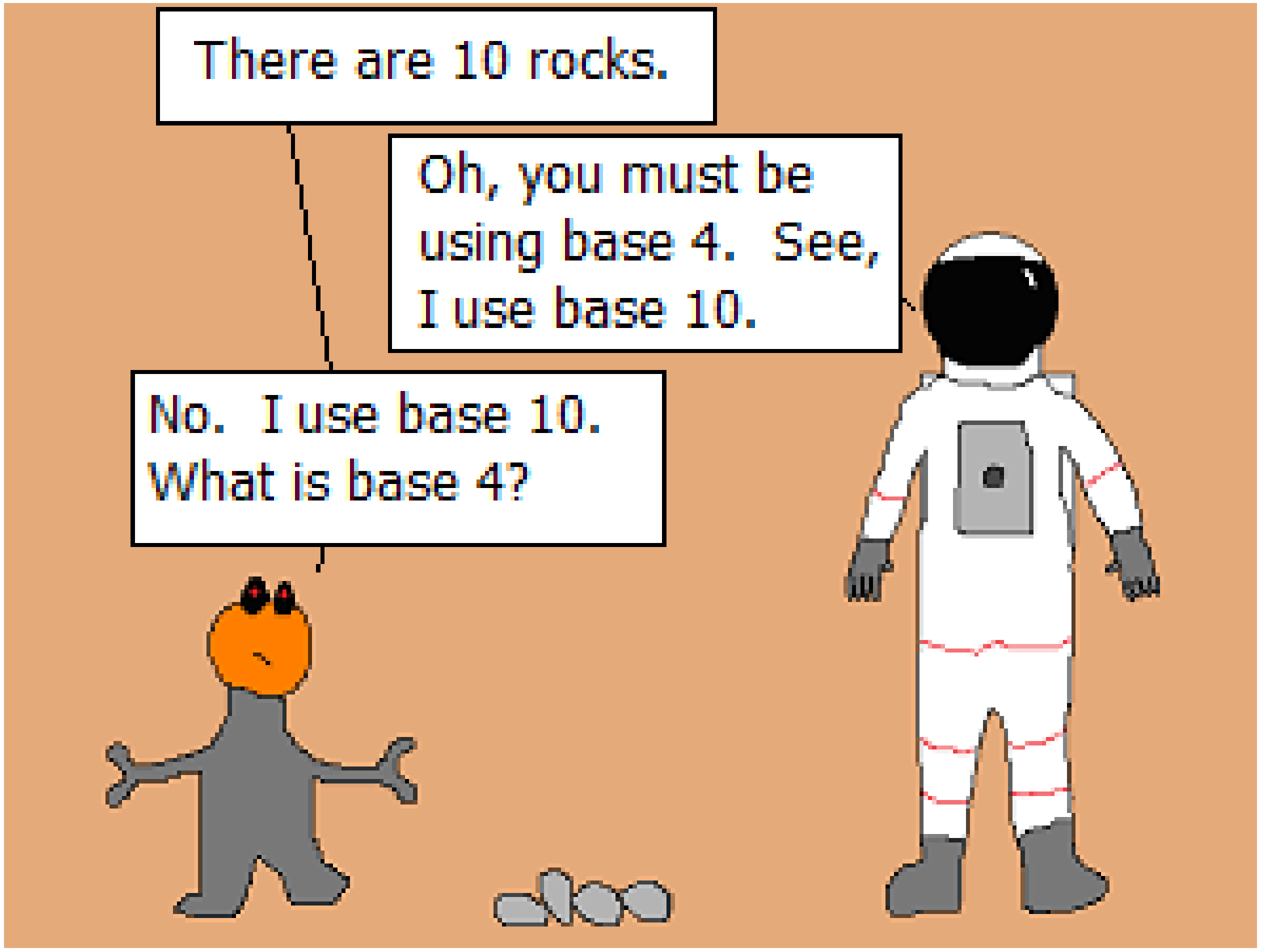
- Regardless of source or destination signed/unsigned type, truncation always just truncates
- This can cause the number to **change drastically in sign and value**

Type promotion in the char/short/int/long family

```
char          sc = 0xFF;    // 0xFF = -1
unsigned char uc = 0xFF;    // 0xFF = 255
int           s1 = sc;      // 0xFFFFFFFF = -1
int           s2 = uc;      // 0x000000FF = 255
unsigned int  u1 = sc;      // 0xFFFFFFFF = 4,294,967,295
unsigned int  u2 = uc;      // 0x000000FF = 255
```

- Promotion always happens according to the *source* variable's type
  - Signed: **"sign extension"** (copy MSB—0 or 1—to fill new space)
  - Unsigned: **"zero fill"** (copy 0's to fill new space)

- *Note:* When doing <, >, <=, >= comparison between different size types, it will promote to the larger type
  - "int < char" comparison will implicitly (1) assign char to int according to these promotion rules, *then* (2) do "int < int" comparison



Every base is base 10.

In closing

Bits As Individual Booleans

THIS IS A VERY DIFFERENT WAY OF THINKING ABOUT WHAT A PARTICULAR SET OF 8 BITS (ONE CHAR) "MEANS"

Bitwise operators and masking

- Let’s say we want to represent font settings:
  - **Bold**
  - *Italic*
  - Red color
  - Superscript
  - Underline
  - ~~Strikethrough~~
- Observe that a particular piece of text can be any combination of these
  - Example 1: ***Bold Italic Red***
  - Example 2: *Italic Red Underline*
  - Example 3: ~~**Bold Superscript Underline Strikethrough**~~

Bitwise operators and masking

- Idea: Have a `bool` for each of these settings, **store them in struct**:

```
struct font_settings {
    bool is_bold;
    bool is_italic;
    bool is_red;
    bool is_super;
    bool is_under;
    bool is_strike;
};
```

This works and is easy to read, but each bool is one byte—7 of each 8 bits not being used. Wastes bigly. Sad!

  - Example 1: ***Bold Italic Red***

```
struct font_settings ex1;    /* how to set up */
ex1.is_bold = ex1.is_italic = ex1.is_red = true;
ex1.is_super = ex1.is_under = ex1.is_strike = false;
if (ex1.is_bold) { …         /* how to use */
```

Bitwise operators and masking

- New idea: Have one 0/1 bit for each of these settings:
  - **Bold** 1 = bold, 0 = not bold
  - *Italic* 1 = italic, 0 = not italic
  - Red color 1 = red, 0 = not red
  - Superscript …
  - Underline
  - ~~Strikethrough~~
- Store the collection of 6 bit settings together:
  - Example 1: ***Bold Italic Red*** `111000`
  - Example 2: *Italic Red Underline* `011010`
  - Example 3: ~~**Bold Superscript Underline Strikethrough**~~ 100111
- **We can pack these into an `unsigned char` (uses lower 6 of the 8 bits)**
  - Example 1: ***Bold Italic Red*** `00111000`

Bitwise operators and masking

- Use char and hexadecimal to store font settings:

  Example 1: ***Bold Italic Red***

  ```
  unsigned char ex1 = 0x38;        // 0x38 = 00111000
  ```

  3 8

- …But how do we use this?

- **No way to "name" the bold bit by itself:**

  ```
  if (ex1) { …                // tests if whole char != 0
  if (ex1.is_bold) { …        // no nameable fields in char
  ```

- **Can't access individual bits (system is byte-addressable)**
- Not hopeless: we need *bitwise operators*

Bitwise operators and bits as individual booleans

MOVING BEYOND THE "INT" INTERPRETATION OF BITS

Bitwise operators

- You've seen these categories of operators in C/C++:
  - Arithmetic operators: `+, -, *, /`
  - Comparison operators: `==, !=, <, >, <=, >=`
  - Logical operators: `&&, ||, !`
  - (C++ only) Stream insertion operators: `<<, >>`
- Now meet a new category:
  - Bitwise operators: `&, |, ^, ~, >>, <<`

shift operators

Bitwise operators

`unsigned char a =`		0	0	1	1	1	1	0	0
`unsigned char b =`		0	1	0	1	1	0	1	0
and, intersection	`a & b`	0	0	0	1	1	0	0	0
or, union	`a | b`	0	1	1	1	1	1	1	0
xor, different?	`a ^ b`	0	1	1	0	0	1	1	0
not	`~a`	1	1	0	0	0	0	1	1
shift left	`a << 2`	1	1	1	1	0	0	0	0
shift right	`a >> 3`	0	0	0	0	0	1	1	1

Bitwise operators and masking

- Use char and hexadecimal to store font settings:

  Example 1: ***Bold Italic Red***

  ```
  unsigned char ex1 = 0x38;        // 0x38 = 00111000
  ```

- **How can we write a test for bold?**

```
bool is_bold(unsigned char settings)
{
    unsigned char mask = 1 << 5;    // 00100000
    return (mask & settings) != 0;
}
```

- **"Mask" is what we call a number that we create solely for the purpose of extracting selected bits out of a bitwise representation**
  - Often crafted using 1 shifted by some amount
  - Writing as a hexadecimal value also acceptable (`0x20`)
  - More complex masks can be crafted in steps with | & etc to test for more than one condition at once

Bitwise operators and masking

- Reminder: here are our font settings, in bit order:
  - **Bold**
  - *Italic*
  - Red color
  - Superscript
  - Underline
  - ~~Strikethrough~~
- **How can we write code to turn off italics (*without* changing any other settings)?**

```
unsigned char italics_off(unsigned char settings)
{
    return ____________________;
}
```

A. `~settings`
B. `settings & (1 << 4)`
C. `settings ^ (1 << 4)`
D. `settings | (~(1 << 4))`
E. `settings & (~(1 << 4))`
F. Something else

(to be) || !(to be), that is the question

- ! and ~ are both "not" operators—are they the same?
- **In other words, is this guaranteed to always print?**

```
int i;
scanf("%d", &i);
if ((!i) == (~i)) printf("same this time\n");
```

A. Yes, always prints
B. Sometimes prints, but not always
C. No, never prints
D. You lost me at the code version of Shakespeare

i = -1 is special case

!(-1) ⇒ 0   !(3) ⇒ 0

~(-1) ⇒ 0000...000   ~(3) ⇒ 111...1100