CIS 2107
Chapter 2 Notes
Part 2

Representing and Manipulating Information

negative numbers?

How do we represent negative numbers?

representing negative numbers

- Sign and magnitude
- One's complement
- Two's complement
- Biased encoding

simple idea: sign and magnitude

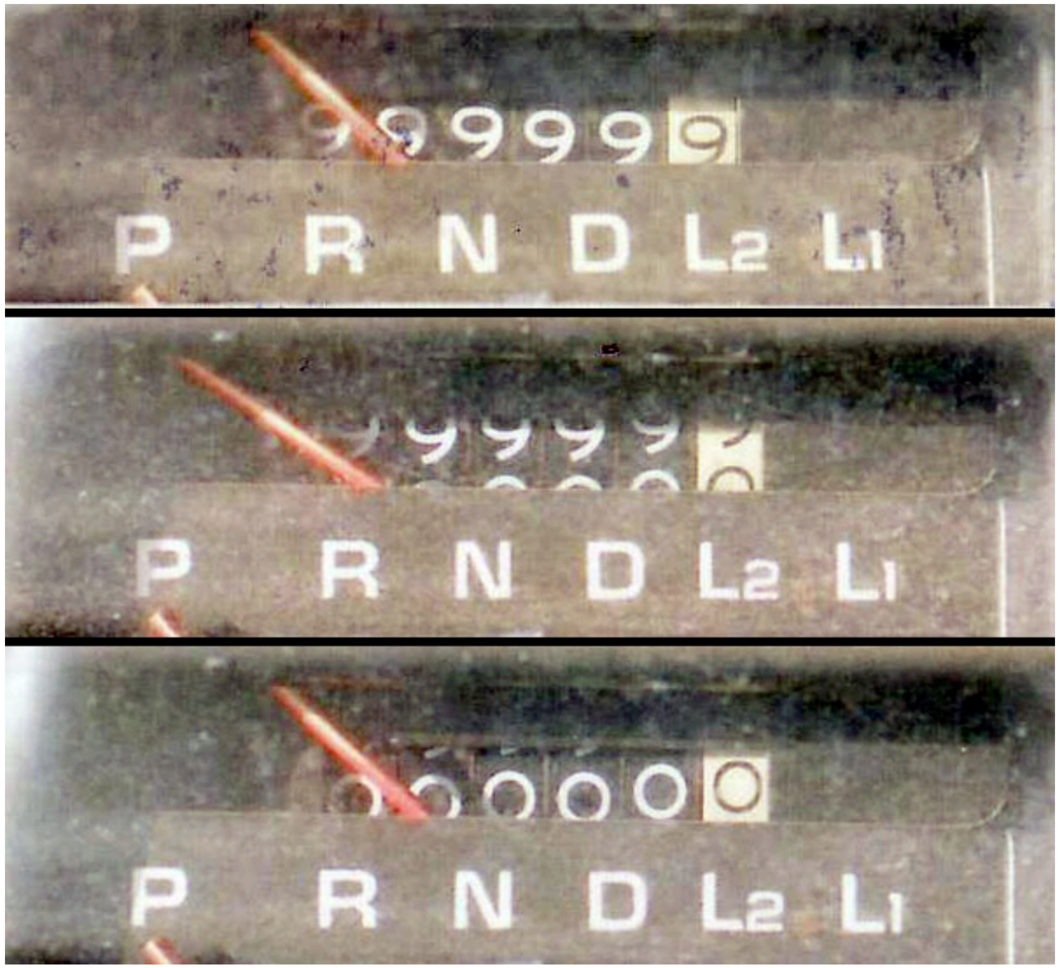
- one bit for the sign
- everything else exactly the same

- So if this is $1_{10:}$
  - $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_2$
- -1_{10} is:
  - $1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_2$

sign and magnitude

- *n* bit numbers, can represent
  - -2^{n-1} to 2^{n-1}
- Some problems:
  - Two zeros:
    - 0_{10}=0x00000000 AND
    - 0_{10}=0x80000000
  - Complicated hardware

another idea: back to our odometer



odometer math

$$
\begin{array}{cccccc}
 & 9 & 9 & 9 & 9 & 9 \\
+ & & & & & 2 \\
\hline
\end{array}
$$

odometer math

$$
\begin{array}{cccccc}
 & 9 & 9 & 9 & 9 & 9 \\
+ & & & & & 2 \\
\hline
\cancel{1} & 0 & 0 & 0 & 0 & 1
\end{array}
$$

$$
\begin{array}{cccccc}
 & 9 & 9 & 9 & 9 & 9 \\
- & 9 & 9 & 9 & 9 & 8 \\
\hline
 & 0 & 0 & 0 & 0 & 1
\end{array}
$$

another example

$$\begin{array}{cccccc} & 9 & 9 & 9 & 9 & 9 \\ + & & 2 & 0 & 0 & 0 \\ \hline \end{array}$$

another example

$$\begin{array}{cccccc} & 9 & 9 & 9 & 9 & 9 \\ + & & 2 & 0 & 0 & 0 \\ \hline \cancel{1} & 0 & 1 & 9 & 9 & 9 \end{array}$$

another example

$$\begin{array}{rrrrrr} & 9 & 9 & 9 & 9 & 9 \\ + & & 2 & 0 & 0 & 0 \\ \hline \cancel{1} & 0 & 1 & 9 & 9 & 9 \end{array}$$

$$\begin{array}{rrrrrr} & 9 & 9 & 9 & 9 & 9 \\ - & 9 & 8 & 0 & 0 & 0 \\ \hline & 0 & 1 & 9 & 9 & 9 \end{array}$$

odometer subtraction?

- (Isn't that illegal?)
- What if we wanted to do subtraction all along?
  - subtracting 99,998 is the same as adding 2
  - subtracting 98,000 is the same as adding 2,000

- In odometer math, to subtract, what can we add?

9’s complement

- decimal 9’s complement of a number:
  - subtract number from all-9’s (same width)
- example: 9’s complement of 31,692

$$\begin{array}{rr} & 99{,}999 \\ - & 31{,}692 \\ \hline & 68{,}307 \end{array}$$

10’s complement

- Simple. 9’s complement + 1.
- 10’s complement of 31,692:

$$
\begin{array}{rr}
 & 99{,}999 \\
- & 31{,}692 \\
\hline
 & 68{,}307 \\
+ & 1 \\
\hline
 & 68{,}308
\end{array}
$$

10's complement. So What?

- In odometer math, adding 68,308 is the same as subtracting 31,692. Example:

$$\begin{array}{rr} & 50{,}000 \\ - & 31{,}692 \\ \hline & 18{,}308 \end{array}$$

$$\begin{array}{rr} & 50{,}000 \\ + & 68{,}308 \\ \hline & 118{,}308 \end{array} \quad \Rightarrow \quad \begin{array}{rr} & 50{,}000 \\ + & 68{,}308 \\ \hline & \cancel{1}18{,}308 \end{array}$$

Why this works

- losing the carry: same as subtracting 100,000

$$
\begin{aligned}
&= 50,000 - 31,692 \\
&= 50,000 + (99,999 - 31,692 + 1) - 100,000 \\
&= 50,000 + (\cancel{99,999} - 31,692 + \cancel{1}) - \cancel{100,000}
\end{aligned}
$$

one's complement

- subtract from an equal-width string of all 1's.
- example: 1's complement of 11000110_2

$$
\begin{array}{rr}
 & 11111111_2 \\
- & 11000110_2 \\
\hline
 & 00111001_2
\end{array}
$$

- another way of getting one's complement?

one's complement

- subtract from an equal-width string of all 1's.
- example: 1's complement of 11000110_2

$$\begin{array}{rr} & 11111111_2 \\ - & 11000110_2 \\ \hline & 00111001_2 \end{array}$$

- another way of getting one's complement?
  - bitwise NOT

one’s complement arithmetic

- Addition
  - add the carry bit back in
- Subtraction
  - add the complement

- notice: like signed magnitude, two zeros

two’s complement

- two’s complement = one’s complement + 1

two's complement example

- 6_{10}=0110_2
- -6_{10}?

- take one's complement (*i.e.*, flip bits), add 1
- OC(0110_2) = 1001_2
- 1001_2+1 = 1010_2
- So -6_{10} = 1010_2

TC(TC(x))=x?

- We've said:
  - 6_{10}=0110_2
  - -6_{10} = 1010_2
- TC(TC(0110_2))=0110_2?
- TC(1010_2)
  - take one's complement
    - 0101_2
  - add one: 0110_2

6_{10}=0110_2; -6_{10} = 1010_2. Check.

- So what's 6 + (-6)?

$$
\begin{array}{rr}
 & 0110_2 \\
+ & 1010_2 \\
\hline
 & \cancel{1}\,0000_2
\end{array}
$$

TC in C

- How could we take the two’s complement of a number in C?

TC in C

- How could we take the two's complement of a number in C?

```
int TC(int x) {
   ??????
}
```

TC in C

- How could we take the two's complement of a number in C? *Easy.*

```
int TC(int x) {
    return -x;
}
```

TC in C

- How could we take the two's complement of a number in C? *Easy.*

```
int TC(int x) {
    return -x;
}
```

- But what if we said that you couldn't use the negation operator?

TC in C without negation operator

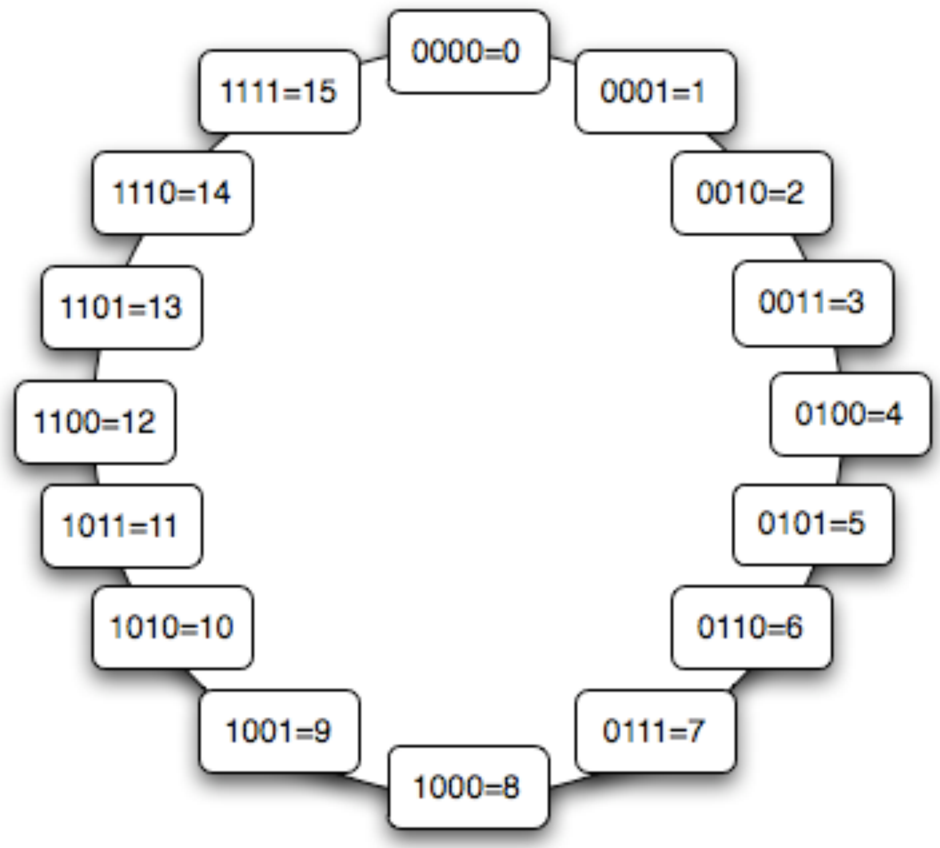
- Take the one's complement (*i.e.*, flip bits)
- Add 1
- How do we do this in C?

TC in C without negation operator

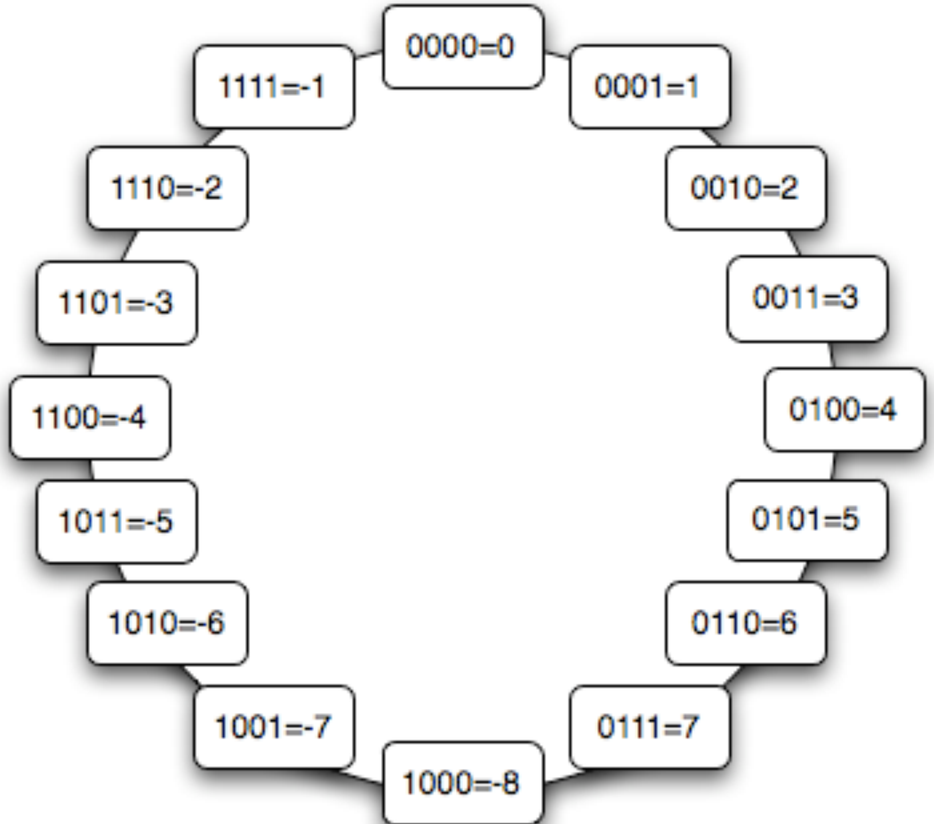
- Take the one's complement (*i.e.*, flip bits)
- Add 1
- How do we do this in C?

```
int TC(int x) {
   return ~x+1;
}
```

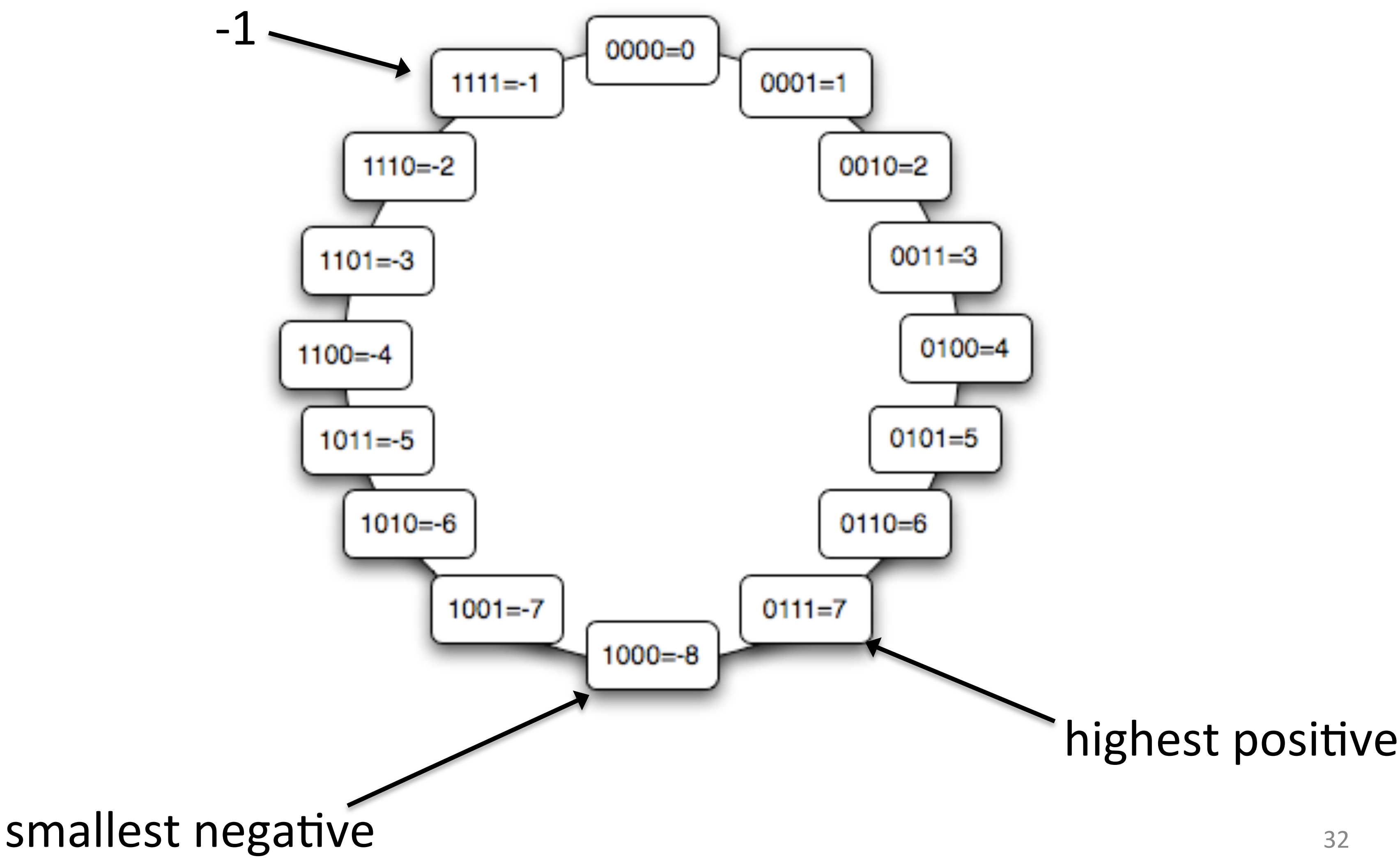
unsigned numbers



signed numbers



know the key places



key places for 16 bit words

	hex	**binary**
UMax	0xFFFF	0b1111111111111111
TCMax	0x7FFF	0b0111111111111111
TCMin	0x8000	0b1000000000000000
0	0x0000	0b0000000000000000
-1	0xFFFF	0b1111111111111111

key places table

word size	8	16	32	64
UMax	0xFF 255_{10}	0xFFFF $65,535_{10}$	0xFFFFFFFF $2^{32}-1$	0xFFFFFFFFFFFFFFFF $2^{64}-1$
TCMax	0x7F	0x7FFF	0x7FFFFFFF	0x7FFFFFFFFFFFFFFF
TCMin	0x80	0x8000	0x80000000	0x8000000000000000
0	0x00	0x0000	0x00000000	0x0000000000000000
-1	0xFF	0xFFFF	0xFFFFFFFF	0xFFFFFFFFFFFFFFF

- know these in binary

unsigned numbers

- another way of describing unsigned nums

$$B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

What's B2U(1010)?

$$B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

$$\begin{aligned} B2U(1010_2) &= (1)(8) + (0)(4) + (1)(2) + (0)(1) \\ &= 10_{10} \end{aligned}$$

same for two's complement

$$B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

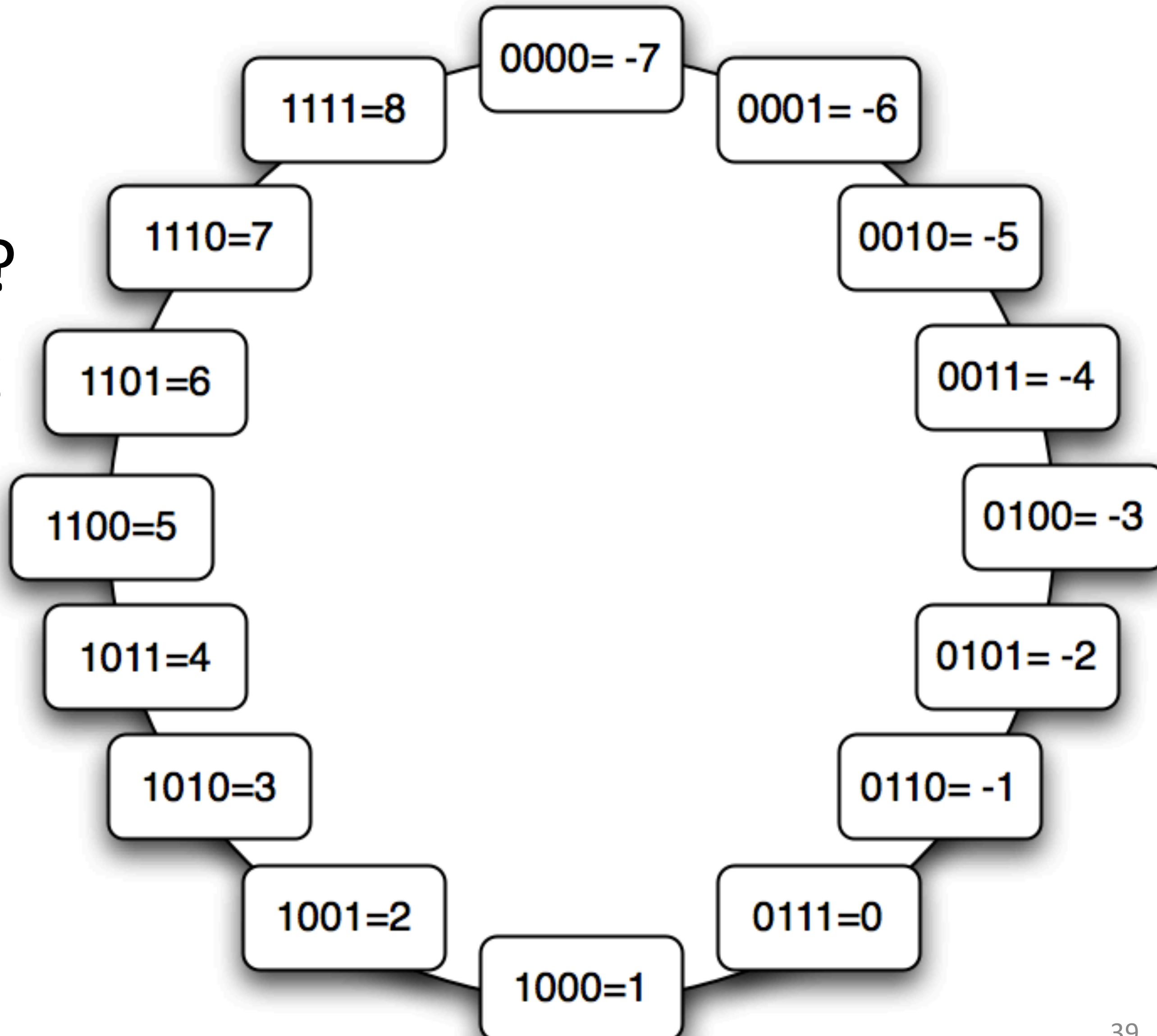
What's TC(1010)?

$$B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

$$\begin{aligned} B2T(1010) &= (1)\mathbf{(\text{-}8)} + (0)(4) + (1)(2) + (0)(1) \\ &= -6 \end{aligned}$$

Bias Encoding. Bias= $-2^{\text{width-1}}-1$

- # zeros?
- # positives?
- order vs TC order



binary	**unsigned**	**TC**	**OC**	**SM**
0000	0	0	0	0
0001	1	1	1	1
0010	2	2	2	2
0011	3	3	3	3
0100	4	4	4	4
0101	5	5	5	5
0110	6	6	6	6
0111	7	7	7	7
1000	8	-8	-7	-0
1001	9	-7	-6	-1
1010	10	-6	-5	-2
1011	11	-5	-4	-3
1100	12	-4	-3	-4
1101	13	-3	-2	-5
1110	14	-2	-1	-6
1111	15	-1	-0	-7

C integer types

- char, short int, int, long int (C99 long long int)
- unsigned versions of each

- C spec doesn't say, but most implementations TC

How big are each?

C data type	typical 32-bit	Intel IA32	x86-64
char	1	1	1
short	2	2	2
int	4	4	4
long	4	4	8
long long	8	8	8
float	4	4	4
double	8	8	8
long double	8	10/12	10/16
pointer	4	4	8

Is this sort of thing an issue in Java?

constants

integers

- 123 (32-bit int)
- 123L (32-bit long int)
- 123u (32-bit unsigned int),
- 123uL (32-bit unsigned long)
- 123LL (64-bit signed long long)

octal and hex

- 0100 (100 octal = 64 decimal)
- 0x100 (100 hex = 256 decimal)
- 0xful (15 unsigned long).

floating point

- 12.3 (32-bit float),
- 123e-1 (32-bit float)
- 12.3f (32-bit float),
- 12.3L (64-bit long double)

'x' = character constant (0 to 127).

- In ASCII ' ' = '\040' = 'x20' = 32
- In ASCII '0' = '\060' = 'x30' = 48
- In ASCII 'A' = '\101' = 'x41' = 65
- In ASCII 'a' = '\141' = 'x61' = 97

What range can be stored?

width	**signed?**	**range**
8 bits	unsigned	0 to $2^8 - 1$
	signed	-2^7 to $2^7 - 1$
16 bits	unsigned	0 to $2^{16} - 1$
	signed	-2^{15} to $2^{15} - 1$
32 bits	unsigned	0 to $2^{32} - 1$
	signed	-2^{31} to $2^{31} - 1$
64 bits	unsigned	0 to $2^{64} - 1$
	signed	-2^{63} to $2^{63} - 1$

limits.h

```
/* Number of bits in a ‘char’.  */
#  define CHAR_BIT      8

/* Minimum and maximum values a ‘signed char’ can hold.  */
#  define SCHAR_MIN     (-128)
#  define SCHAR_MAX     127

/* Maximum value an ‘unsigned char’ can hold.  (Minimum is 0.)  */
#  define UCHAR_MAX     255

/* Minimum and maximum values a ‘signed short int’ can hold.  */
#  define SHRT_MIN      (-32768)
#  define SHRT_MAX      32767

/* Maximum value an ‘unsigned short int’ can hold.  (Minimum is 0.)  */
#  define USHRT_MAX     65535

/* Minimum and maximum values a ‘signed int’ can hold.  */
#  define INT_MIN       (-INT_MAX - 1)
#  define INT_MAX       2147483647

/* Maximum value an ‘unsigned int’ can hold.  (Minimum is 0.)  */
#  define UINT_MAX      4294967295U

/* Minimum and maximum values a ‘signed long int’ can hold.  */
#  define LONG_MAX     2147483647L
#  define LONG_MIN      (-LONG_MAX - 1L)
```

C99 <stdint.h>

- for all widths *W* that the machine supports
  - exact width types: int*W*_t, uint*W*_t
    - *e.g.*, int8_t, uint32_t.
  - minimum width types: int_least*W*_t, uint_least*W*_t
  - <limits.h>-style macros for these types:
    - e.g. INT*W*_MIN, UINT*W*_MAX, …

take a look at some of these with gdb

```
int p1 = 37;
unsigned int p2 = 37;
int n1 = -37;
```

- remember to compile:
  - with the –g switch
  - don't turn on optimization (*no* –O)
- in gdb, to print binary, use p/t
- confirm you get the expected when you try:
  - p/d (~n1+1)

What happens?

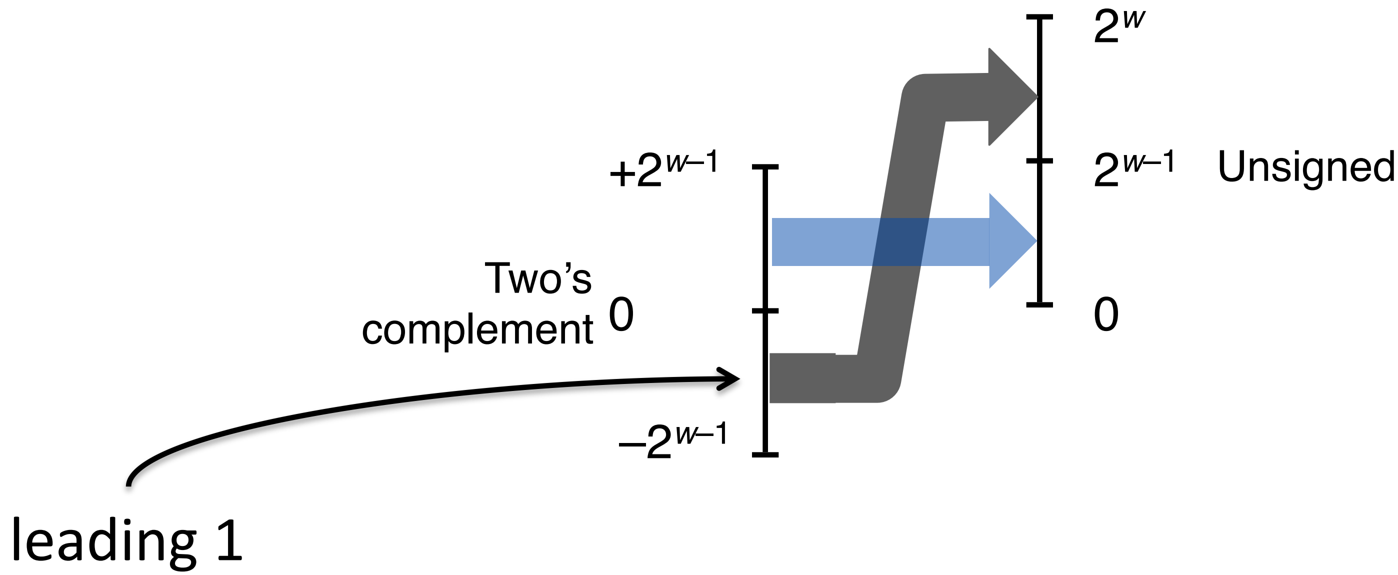
```
int x=-1;
unsigned int ux = (unsigned int) x;
```

What happens?

```
int x=-1;
unsigned int ux = (unsigned int) x;


(gdb) p/t x
$1 = 11111111111111111111111111111111
(gdb) p/d x
$2 = -1
(gdb) p/t ux
$3 = 11111111111111111111111111111111
(gdb) p/u ux
$4 = 4294967295
```

casting from signed to unsigned

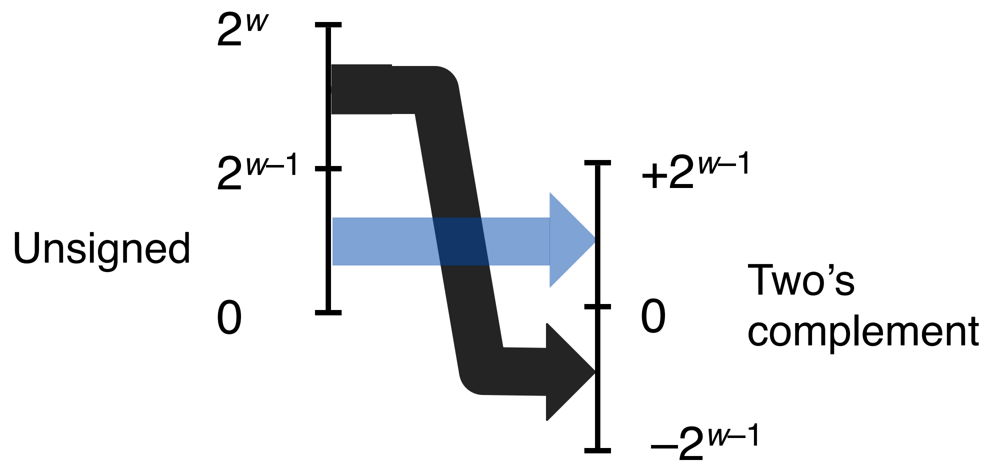


what happens if we cast a negative to an equal width unsigned?

bits	**signed**	**unsigned**
0000_2	00_{10}	00_{10}
0001_2	01_{10}	01_{10}
0010_2	02_{10}	02_{10}
0011_2	03_{10}	03_{10}
0100_2	04_{10}	04_{10}
0101_2	05_{10}	05_{10}
0110_2	06_{10}	06_{10}
0111_2	07_{10}	07_{10}
1000_2	-08_{10}	08_{10}
1001_2	-07_{10}	09_{10}
1010_2	-06_{10}	10_{10}
1011_2	-05_{10}	11_{10}
1100_2	-04_{10}	12_{10}
1101_2	-03_{10}	13_{10}
1110_2	-02_{10}	14_{10}
1111_2	-01_{10}	15_{10}

bits	signed		unsigned
0000_2	00_{10}		00_{10}
0001_2	01_{10}		01_{10}
0010_2	02_{10}		02_{10}
0011_2	03_{10}	same	03_{10}
0100_2	04_{10}		04_{10}
0101_2	05_{10}		05_{10}
0110_2	06_{10}		06_{10}
0111_2	07_{10}		07_{10}
1000_2	-08_{10}		08_{10}
1001_2	-07_{10}		09_{10}
1010_2	-06_{10}		10_{10}
1011_2	-05_{10}	+16	11_{10}
1100_2	-04_{10}		12_{10}
1101_2	-03_{10}		13_{10}
1110_2	-02_{10}		14_{10}
1111_2	-01_{10}		15_{10}

casting from unsigned to signed



what if we cast a large unsigned positive to an equal width signed?

bit shifting for unsigned numbers

- >>, <<, >>=, <<=
- numbers fall off the end
- fill in with 0s
- math equivalent (when 1s haven't fallen off)?

0	0	1	0	1	1	0	1

shift left 1

0	1	0	1	1	0	1	0

shift right 2

0	0	0	1	0	1	1	0

shifting left

- *x* << *j*
  - shift *x* to the left *j* bit positions
  - fill with 0s from the right
  - numbers “fall off the left end”

```
char x=11, j;

for (j=0; j<8; j++)
  x<<=1;
```

	x_{10}	x_2
0	11	0b00001011
1	22	0b00010110
2	44	0b00101100
3	88	0b01011000
4	176	0b10110000
5	96	0b01100000
6	192	0b11000000
7	128	0b10000000

shifting to the right

- unsigned:
  - same as left shift
- what about signed?
  - implementation dependent
  - some fill from LHS
    - with 0
    - with sign bit
  - (why do this with signed numbers anyway?)

```
int main(void)
{
  int i;
  char c1 = 64, c2 = -64;
  for (i=8; i>0; i--) {
    c1>>=1;
    c2>>=1;
  }
  return 0;
}
```

(aside) java right shift

- Java defines two:
  - **>>** fills from the left with the sign bit
  - **>>>** fills from the left with 0s

some example code

```
public class JavaShift {
  public static void main(String args[]) {
    int x = Integer.MIN_VALUE; // i.e. -2**(31)
    for (int i=0; i<32; i++) {
      int ds = x>>i;
      int ts = x>>>i;
      System.out.println(x+" >> "+ i + " = " +
                                   Integer.toBinaryString(ds));
      System.out.println(x+" >>> "+ i + " = " +
                                   Integer.toBinaryString(ts));
      System.out.println();
    }
  }
}
```

java right shifts output

```
-2147483648 >> 0 = 10000000000000000000000000000000
-2147483648 >>> 0 = 10000000000000000000000000000000

-2147483648 >> 1 = 11000000000000000000000000000000
-2147483648 >>> 1 = 1000000000000000000000000000000

-2147483648 >> 2 = 11100000000000000000000000000000
-2147483648 >>> 2 = 100000000000000000000000000000

-2147483648 >> 3 = 11110000000000000000000000000000
-2147483648 >>> 3 = 10000000000000000000000000000

-2147483648 >> 4 = 11111000000000000000000000000000
-2147483648 >>> 4 = 1000000000000000000000000000

-2147483648 >> 5 = 11111100000000000000000000000000
-2147483648 >>> 5 = 100000000000000000000000000
```

casting to different widths

- smaller to larger
  - no problem
- larger to smaller
  - information loss?

small to large

- What happens?
  - long x = char c

unsigned: zero extension

- unsigned numbers
  - going from small to large → zero extension
- Example: unsigned long x = unsigned char c
- What happens?

signed: sign extension

- signed numbers
  - going from small to large → sign extension
- Example: long x = char c
- What happens?
  - if c > 0?
  - if c < 0?

C conversion rules

- Details in K&R Appendix A
- C rules. Three types:
  - Integer promotion
  - Integer conversion rank
  - Usual arithmetic conversions

C Conversion Rules: Integer Promotion

- Integer types smaller than int promoted to int.
- example:
  - char result, a=100, b=10, c=20

C Conversion Rules: Integer Conversion Rank

C Conversion Rules: "Usual" Arithmetic Conversions

casting large to small. truncation

```
int x1 = 0x00001234;
int x2 = 0x12345678;
short s1 = (short)x1;
short s2 = (short)x2;
printf("x1=0x%08x, ", x1);
printf("x2=0x%08x, ", x2);
printf("s1=0x%08x, ", s1);
printf("s2=0x%08x\n", s2);
```

we get:

```
x1=0x00001234, x2=0x12345678, s1=0x00001234, s2=0x00005678
```

truncation of unsigned numbers

- unsigned int x
- truncating to *k* bits equivalent to x mod 2^k

integer arithmetic

- “odometer” effect
- modular arithmetic
- unsigned addition: addition modulo width
- for others, see details in BO

unsigned addition

- addition modulo the width

@@@ TAKE A LOOK AT BO SLIDES 28 --@@@

@@@ ADD CMU SL 31 MATERIAL @@@

Floating Point

What happens here?

```c
#include <stdio.h>

int main(int argc, char **argv)
{
  float f=0.1;

  if (f==0.1)
    printf("It’s 0.1\n");
  else
    printf("It’s not 0.1\n");

  return 0;
}
```

What happens here?

```
#include <stdio.h>

int main(int argc, char **argv)
{
  float f=0.1;

  if (f==0.1)
    printf("It's 0.1\n");
  else
    printf("It's not 0.1\n");

  return 0;
}
```

```
bash-3.2$ gcc -Wall -o StrangeFloat02 StrangeFloat02.c
bash-3.2$ ./StrangeFloat02
It's not 0.1
```

What about here?

```c
#include <stdio.h>

int main(int argc, char **argv)
{
  float f1=0.1,
    f2=0.2,
    sum;

  sum=f1+f2;
  printf("%.8f+%.8f=%.8f\n", f1, f2, sum);

  return 0;
}
```

What about here?

```c
#include <stdio.h>

int main(int argc, char **argv)
{
  float f1=0.1,
    f2=0.2,
    sum;

  sum=f1+f2;
  printf("%.8f+%.8f=%.8f\n", f1, f2, sum);

  return 0;
}
```

```
bash-3.2$ gcc -Wall -o StrangeFloat StrangeFloat.c
bash-3.2$ ./StrangeFloat
0.10000000+0.20000000=0.30000001
```

Strange on 32-bit machines

```
#include <stdio.h>

int main(void) {
  float a = 30000000;
  float b = 3;
  float  c;

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

  return 0;
}
```

Strange on 32-bit machines

```c
#include <stdio.h>

int main(void) {
  float a = 30000000;
  float b = 3;
  float  c;

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

  return 0;
}
```

Output:

```
3.000000
4.000000
```

Recall Binary Representation of ints

$$11011001_2 =$$

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	1	0	1	1	0	0	1

$$
\begin{aligned}
&= (1)(128) + (1)(64) + (0)(32) + (1)(16) + (1)(8) + (0)(4) + (0)(2) + (1)(1) \\
&= 128 + 64 + 16 + 8 + 1 \\
&= 217
\end{aligned}
$$

What about fractions?

$$.11011001_2 =$$

.	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}
.	1	1	0	1	1	0	0	1

$$= 2^{-1} + 2^{-2} + 2^{-4} + 2^{-5} + 2^{-8}$$

$$= 0.5 + 0.25 + 0.0625 + 0.03125 + 0.00390625$$

$$= 0.84765625$$

some powers of 2 <= 1

2^{0}	1.000000
2^{-1}	0.500000
2^{-2}	0.250000
2^{-3}	0.125000
2^{-4}	0.062500
2^{-5}	0.031250
2^{-6}	0.015625
2^{-7}	0.0078125
2^{-8}	0.00390625

Decimal to binary?

- How do we convert the representation of:
  - a fraction in decimal
  - to a fraction in binary

recall how we did numbers > 1

- Convert 119_{10} to binary
- solution: $111\ 0111_2$

$$119 = 59 * 2 + \mathbf{1}$$
$$59 = 29 * 2 + \mathbf{1}$$
$$29 = 14 * 2 + \mathbf{1}$$
$$14 = 7 * 2 + \mathbf{0}$$
$$7 = 3 * 2 + \mathbf{1}$$
$$3 = 1 * 2 + \mathbf{1}$$
$$1 = 0 * 2 + \mathbf{1}$$

converting fractions

```
while (fraction part != 0) {
  multiply number by 2
  record the integer part for later
  subtract the integer part
}

recorded integer parts are the binary rep
```

example. 0.6953125_{10}?

$$0.6953125_{10} * 2 = \mathbf{1}.390625_{10}$$

$$0.390625_{10} * 2 = \mathbf{0}.78125_{10}$$

$$0.78125_{10} * 2 = \mathbf{1}.5625_{10}$$

$$0.5625_{10} * 2 = \mathbf{1}.125_{10}$$

$$0.125_{10} * 2 = \mathbf{0}.25_{10}$$

$$0.25_{10} * 2 = \mathbf{0}.5_{10}$$

$$0.5_{10} * 2 = \mathbf{1}.0_{10}$$

solution: $0.6953125_{10}=0.1011001_2$

double check result

- $0.1011001_2 = 0.6953125_{10}$????

$$
\begin{aligned}
0.1011001_2 &= 2^{-1} + 2^{-3} + 2^{-4} + 2^{-7} \\
&= 0.5_{10} + 0.125_{10} + 0.0625_{10} + 0.0078125_{10}
\end{aligned}
$$

$$
\begin{array}{rl}
 & 0.5_{10} \\
 & 0.125_{10} \\
 & 0.0625_{10} \\
+ & 0.0078125_{10} \\
\hline
 & 0.6953125_{10}
\end{array}
$$

Does it always work?

- Remember the pigeonhole?
  - infinite number of floating-point numbers
  - storing in register of finite size

Does it always work? Repeating

- We have repeating decimal fractions
- We also have repeating binary fractions.
  - $0.1_{10} = 0.0001100110011_2 \ldots$
  - $0.2_{10} = 0.001100110011_2 \ldots$
- The more places we have, the closer we are to the value we're trying to represent

representing 0.2_{10}

base 2 float	**base 10 frac**	**base 10 float**
0.0_2	0_{10}	0_{10}
0.00_2	0_{10}	0_{10}
0.001_2	$1/8_{10}$	0.125_{10}
0.0011_2	$3/16_{10}$	0.1875_{10}
0.00110_2	$3/16_{10}$	0.1875_{10}
0.001100_2	$3/16_{10}$	0.1875_{10}
0.0011001_2	$25/128_{10}$	0.1953125_{10}
0.00110011_2	$51/256_{10}$	0.19921875_{10}
0.001100110_2	$51/256_{10}$	0.19921875_{10}
0.0011001100_2	$51/256_{10}$	0.19921875_{10}
0.00110011001_2	$409/2048_{10}$	0.19970703125_{10}
0.001100110011_2	$819/4096_{10}$	0.199951171875_{10}
0.0011001100110_2	$819/4096_{10}$	0.199951171875_{10}
0.00110011001100_2	$819/4096_{10}$	0.199951171875_{10}
0.001100110011001_2	$6553/32768_{10}$	0.199981689453125_{10}
0.0011001100110011_2	$13107/65536_{10}$	0.1999969482421875_{10}
0.00110011001100110_2	$13107/65536_{10}$	0.1999969482421875_{10}
0.001100110011001100_2	$13107/65536_{10}$	0.1999969482421875_{10}
0.0011001100110011001_2	$104857/524288_{10}$	$0.1999988555908203125_{10}$
0.00110011001100110011_2	$209715/1048576_{10}$	$0.19999980926513671875_{10}$

representing 0.1_{10}

base 2 float	base 10 frac	base 10 float
0.00_2	0_{10}	0_{10}
0.000_2	0_{10}	0_{10}
0.0001_2	$1/16_{10}$	0.0625_{10}
0.00011_2	$3/32_{10}$	0.09375_{10}
0.000110_2	$3/32_{10}$	0.09375_{10}
0.0001100_2	$3/32_{10}$	0.09375_{10}
0.00011001_2	$25/256_{10}$	0.09765625_{10}
0.000110011_2	$51/512_{10}$	0.099609375_{10}
0.0001100110_2	$51/512_{10}$	0.099609375_{10}
0.00011001100_2	$51/512_{10}$	0.099609375_{10}
0.000110011001_2	$409/4096_{10}$	0.099853515625_{10}
0.0001100110011_2	$819/8192_{10}$	0.0999755859375_{10}
0.00011001100110_2	$819/8192_{10}$	0.0999755859375_{10}
0.000110011001100_2	$819/8192_{10}$	0.0999755859375_{10}
0.0001100110011001_2	$6553/65536_{10}$	0.0999908447265625_{10}
0.00011001100110011_2	$13107/131072_{10}$	0.09999847412109375_{10}
0.000110011001100110_2	$13107/131072_{10}$	0.09999847412109375_{10}
0.0001100110011001100_2	$13107/131072_{10}$	0.09999847412109375_{10}
0.00011001100110011001_2	$104857/1048576_{10}$	$0.09999942779541015625_{10}$
$0.000110011001100110011_2$	$209715/2097152_{10}$	$0.099999904632568359375_{10}$

can't represent everything

- with finite length binary strings, can only approximate numbers that can't be written as:

$$(x)(2^y)$$

- (remember that *y* can be positive or negative)

Other "interesting" numbers

base 2 float	**base 10 frac**	**base 10 float**
0.1_2	$1/2_{10}$	0.5_{10}
0.11_2	$3/4_{10}$	0.75_{10}
0.111_2	$7/8_{10}$	0.875_{10}
0.1111_2	$15/16_{10}$	0.9375_{10}
0.11111_2	$31/32_{10}$	0.96875_{10}
0.111111_2	$63/64_{10}$	0.984375_{10}
0.1111111_2	$127/128_{10}$	0.9921875_{10}
0.11111111_2	$255/256_{10}$	0.99609375_{10}
0.111111111_2	$511/512_{10}$	0.998046875_{10}
0.1111111111_2	$1023/1024_{10}$	0.9990234375_{10}

machine representation of floats

- represent as:

$$\pm m \times b^e$$

$$\begin{array}{ll} m & \text{mantissa} \\ b & \text{base} \\ e & \text{exponent} \end{array}$$

- but, base is always 2

machine representation of floats

- Recall: 1234567.0_{10} can be written:
  - $123456.7 * 10$
  - $12345.67 * 10^2$
  - $1234.567 * 10^3$
  - $123.4567 * 10^4$
  - $12.34567 * 10^5$
  - $1.234567 * 10^6$

machine representation of floats

- 110010.0010_2 can be written:
  - $110010.0010_2 * 2^0$
  - $11001.00010_2 * 2^1$
  - $1100.100010_2 * 2^2$
  - $110.0100010_2 * 2^3$
  - $11.00100010_2 * 2^4$
  - *$1.100100010_2 * 2^5$*
- Shift until radix after first 1 called *normalizing*

machine representation of floats

floating point representation

C type	exponent	mantissa
`float`	8 bits	23 bits
`double`	11 bits	52 bits

floating point representation

C type	**exponent**	**mantissa**
`float`	8 bits	23 bits
`double`	11 bits	52 bits

On Intel – "extended precision" (depending on compiler)

C type	**exponent**	**mantissa**
`long double`	15 bits	64 bits

How?

- How are floating-point numbers represented?
- There are three cases.

--- let's do the most common case first

How? The common case.

- Set the sign bit. 0 for positive, 1 for negative
- Write num in fixed-pt binary
- Normalize (radix pt is just to the right of the first 1)
- *m* is the values to the right of radix point
- calculate *e*: exponent+*bias*
  - *bias* = $2^{\textit{exponent field width}-1}-1$
  - for floats, bias is $2^{8-1}-1 = 2^7-1 = 127$
  - for doubles, it's $2^{11-1}-1 = 2^{10}-1= 1023$
  - Can represent exponents of:
    - -126 to +127 for floats
    - -1022 to +1023 for doubles

How? Example -15.375

- Set the sign bit. 0 for positive, 1 for negative
  - sign bit 1
- Write num in fixed-pt binary:
  - $15.375 = 1111.011_2$
- Normalize (so radix point is just to the right of the first 1)
  - $1.111011_2 * 2^3$
- *m* is the values to the right of radix point
  - 111011
- calculate *e*: exponent+127
  - exponent was 3_{10} (11_2).
  - $3_{10}+127_{10} = 130_{10}$ or $1000\ 0010_2$
- Final result:

1	1000 0010	111 0110 0000 0000 0000 0000

Another example

- -118.625

Another example

- -118.625

1	1000 0101	110 1101 0100 0000 0000 0000

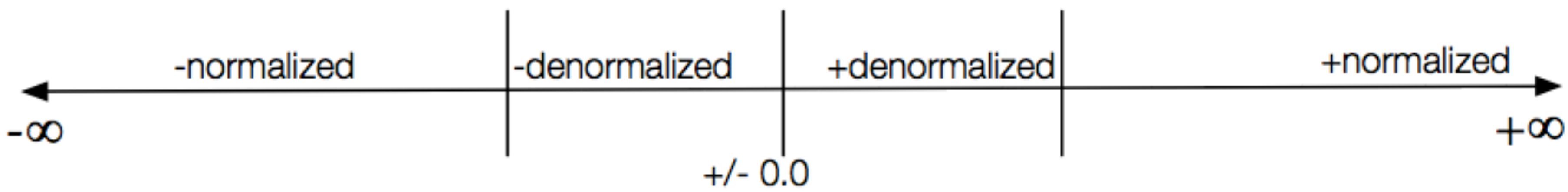
double check: try in gdb

- -118.625? | 1 | 1000 0101 | 110 1101 0100 0000 0000 0000 |
- Check.
- In a program where we have:
  - `float fl`
- `(gdb) set fl=-118.625`
- `(gdb) x/t &fl`
- `0x7fff5fbff68c: 11000010111011010100000000000000`

How?

- How are floating-point numbers represented?
- There are three cases:
  - *Normalized* values (what we just did)
  - *De-normalized* values
    - Numbers “close” to 0.0
    - Exponent field is all zeros
  - “Special” cases
    - +/- ∞, NaN
    - Exponent field is all 1s.

Visualizing Floating-point Range



How?

- How are floating-point numbers represented?
- There are three cases:
  - *Normalized* values (what we just did)
  - ***De-normalized* values**
    - Numbers “close” to 0.0
    - Exponent field is all zeros
  - “Special” cases
    - +/- ∞, NaN
    - Exponent field is all 1s.

De-Normalized Values

- Why?
  - Used to represent values “close” to 0.0
- *Exponent field* – all 0s.
  - Fraction represented has exponent of 1-Bias
- *Mantissa field*
  - we don’t assume a leading 1
- *Sign field*
  - As usual, can be 1 or 0.
  - Means that we can also have +0.0 or -0.0

Example

- We'll use 7-bit floats, consisting of:
  - A sign bit
  - 3 bits for the exponent
  - 3 bits for the mantissa
- What is: 0 101 000, where:
  - 0 is the sign bit
  - 101 are the bits for the mantissa
  - 000 are the bits for the exponent

Example: 0 101 000

- We'll use 7-bit floats, consisting of:
  - A sign bit
  - 3 bits for the exponent
  - 3 bits for the mantissa
- What is: 0 101 000, where:
  - 0 is the sign bit
  - 101 are the bits for the mantissa
  - 000 are the bits for the exponent
- Bias is $2^{(3-1)}-1 = 2^2-1 = 3$
- Exponent is 1-bias = -2
- Mantissa:
  - $½+0/4+1/8 = 5/8$
- Final result:
  - Mantissa * $2^{exponent}$=
  - $5/8*2^{-2}$=
  - 5/8*1/4=
  - 5/32=
  - 0.15625

Another de-normalized example

- 0 110 000 (110 is the mantissa)

Another de-normalized example

- 0 110 000 (110 is the mantissa)
- Bias is $2^{(3-1)}-1 = 2^2-1 = 3$
- Exponent is 1-bias = -2
- M = ½+1/4+0/8 = ¾
- Final result = $M*2^{exponent}$ =
  - $¾*2^{-2}$=
  - 3/4*1/4=
  - 3/16 = 0.1875

How?

- How are floating-point numbers represented?
- There are three cases:
  - *Normalized* values (what we just did)
  - *De-normalized* values
    - Numbers “close” to 0.0
    - Exponent field is all zeros
  - **“Special” cases**
    - +/- ∞, NaN
    - Exponent field is all 1s.

“Special” Cases

- The exponent field is all 1s.
- If the fraction field is all 0s:
  - +/- infinity
  - Can use +/- infinity when:
    - overflow has occurred
    - divide by 0.
- Otherwise:
  - NaN, *e.g.* sqrt(-1)

“Simple” examples

- 32-, 64-, or 80-bit widths: tough to see
- keep it simple for now: 8-bit widths.
  - 1 bit for sign
  - 4 bits for the exponent
  - 3 bits for the fraction

Bias

- With a 4-bit exponent, what will be the bias?

Bias

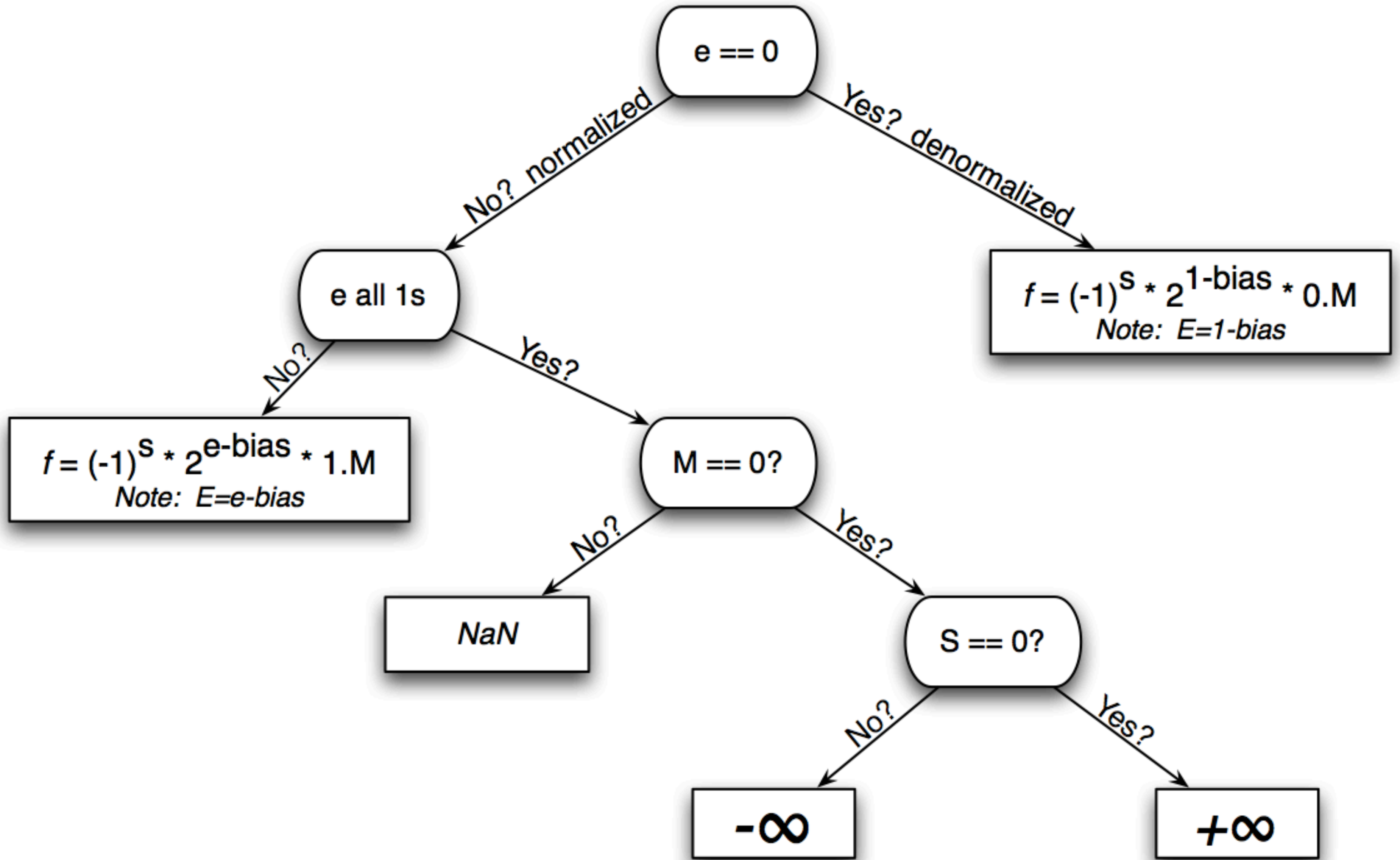
- With a 4-bit exponent, what will be the bias?
- Bias = $2^{width-1}-1 = 2^{4-1}-1 = 2^3-1 = 7$

Table of "Simple" Values

s	exp	frac	E	Value	comment
0	0000	000	-6	0	zero
0	0000	001	-6	$1/8 * 1/64 = 1/512$	closest to zero
0	0000	010	-6	$2/8 * 1/64 = 2/512$	
...	...	...	...	...	
0	0000	110	-6	$6/8 * 1/64 = 6/512$	
0	0000	111	-6	$7/8 * 1/64 = 7/512$	largest denormalized
0	0001	000	-6	$8/8 * 1/64 = 8/512$	smallest norm (rem. leading 1)
0	0001	001	-6	$9/8 * 1/64 = 9/512$	
...	...	...	...	...	...
0	0110	110	-1	$14/8 * 1/2 = 14/16$	
0	0110	111	-1	$15/8 * 1/2 = 15/16$	closest to 1 from below
0	0111	000	0	$8/8 * 1 = 1$	
0	0111	001	0	$9/8 * 1 = 9/8$	closest to 1 from above
0	0111	010	0	$10/8 * 1 = 10/8$	
...	...	...	...	...	...
0	1110	110	7	$14/8 * 128 = 224$	
0	1110	111	7	$15/8 * 128 = 240$	largest normalized
0	1111	000	n/a	infinity	

- Please convince yourselves that these make sense

Floating-point cheat sheet



“important” numbers

description	**exp**	**frac**
zero	00...00	00...00
smallest de-normalized	00...00	00...01
largest de-normalized	00...00	11...11
smallest normalized	00...01	00...00
one	01...11	00...00
largest normalized	11...10	11...11

Rounding

	\$1.40	\$1.60	\$1.50	\$2.50	-\$1.50
towards zero	\$1	\$1	\$1	\$2	-\$1
round down (-∞)	\$1	\$1	\$1	\$2	-\$2
round up (+∞)	\$2	\$2	\$2	\$3	-\$1
round to even	\$1	\$2	\$2	\$2	-\$2

Rounding

	\$1.40	\$1.60	\$1.50	\$2.50	-\$1.50
towards zero	\$1	\$1	\$1	\$2	-\$1
round down (-∞)	\$1	\$1	\$1	\$2	-\$2
round up (+∞)	\$2	\$2	\$2	\$3	-\$1
round to even	\$1	\$2	\$2	\$2	-\$2

- Don't be confused by round to even:
  - Round to the closest
  - Choose the even number when you're half-way between two possibilities

Why Round to Even?

- Rounding in same direction → skew?
- Round to even:
  - sometimes up
  - sometimes down

The moral of the story … back to our first example. Why false?

```c
#include <stdio.h>

int main(int argc, char **argv)
{
  float f=0.1;

  if (f==0.1)
    printf("It’s 0.1\n");
  else
    printf("It’s not 0.1\n");

  return 0;
}
```

The moral of the story … back to our first example. Why false?

```
1  #include <stdio.h>
2
3  int main(int argc, char **argv)
4  {
5    float f=0.1;
6
7    if (f==0.1)
8      printf("It's 0.1\n");
9    else
10     printf("It's not 0.1\n");
11
12   return 0;
13 }
```

- Can we store 0.1_{10} without rounding?
- Will there be less round error if we use a double?
- What happens if the compiler uses a double for the 0.1 in line 7? Will the values be the same?

These give us what we expect.

```c
#include <stdio.h>

int main(int argc, char **argv)
{
  float f = 0.1;

  if (f==0.1f) {
    printf("It's 0.1\n");
  } else {
    printf("It's not 0.1\n");
  }

  return 0;
}
```

```c
#include <stdio.h>

int main(int argc, char **argv)
{
  double f = 0.1;

  if (f==0.1) {
    printf("It's 0.1\n");
  } else {
    printf("It's not 0.1\n");
  }

  return 0;
}
```