

CSE 374: Lecture 22

Hexadecimal and number storage



Number systems and BASE

Generally use base 10

(10 fingers)

234

$2 \times 100 + 3 \times 10 + 4 \times 1$

$2 \times 10^2 + 3 \times 10^1 + 4 \times 10^0$

Digital systems - base 2

(binary)

$234 = 0b11101010$

**$1 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 +$
 $1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$**

Base 16 - very compact

(hexadecimal)

$234 = 0xEA$

$14 \times 16^1 + 10 \times 16^0$

**Need 16 digits,
so we used [0-9A-F]**

Notice: 234 takes 3 digits to express in base 10, 8 in base 2, and 2 in base 16.

Integer representations

Digital systems are 'on' or 'off', thus, Binary.

→ The hardware (and C) supports two flavors of integers

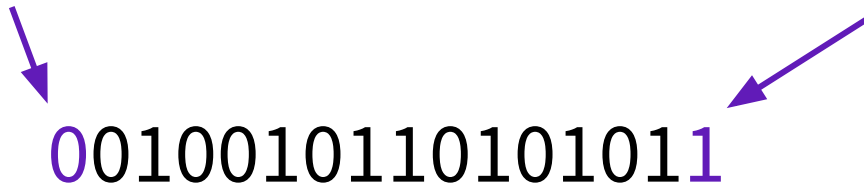
- ◆ unsigned – only the non-negatives
- ◆ signed – both negatives and non-negatives

→ There are only 2^W distinct bit patterns of W bits, so...

- ◆ Cannot represent all the integers
- ◆ Unsigned values: $0 \dots 2^W - 1$ $\leftarrow 2^4 - 1 \rightarrow 1111 \rightarrow 2^3 + 2^2 + 2^1 + 2^0 \rightarrow 8 + 4 + 2 + 1 \rightarrow 15$
- ◆ Signed values: $-2^{W-1} \dots 2^{W-1} - 1$

→ Reminder: terminology for binary representations

“Most-significant” / “high-order” bit(s) “Least-significant” / “low-order” bit(s)

The diagram shows the binary number 0010010110101011. A purple arrow points from the text "Most-significant" / "high-order" bit(s) to the first '0' on the left. Another purple arrow points from the text "Least-significant" / "low-order" bit(s) to the last '1' on the right.

0010010110101011

Signed Ints (obvious solution)

4 bit signed int

Most significant bit is reserved for the sign

Changes the range to $[-2^{w-1}-1, 2^{w-1}-1]$

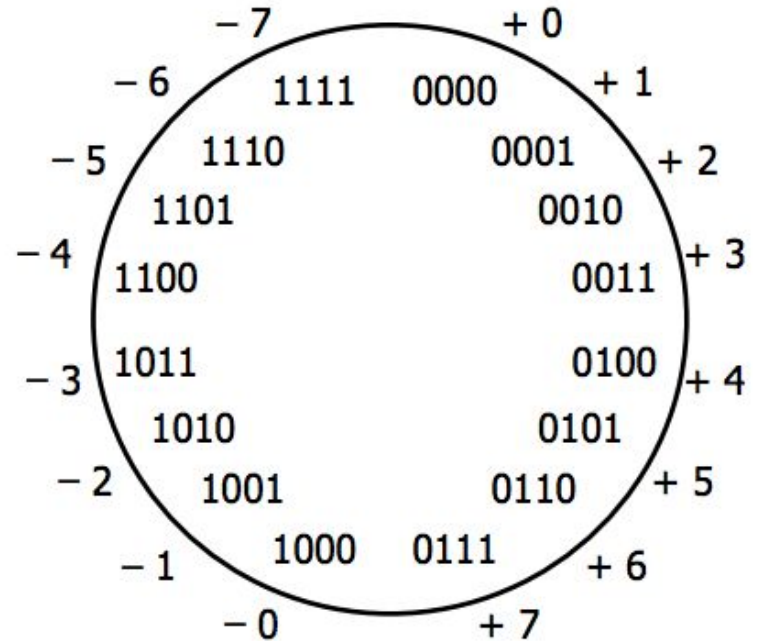
Adding unsigned ints :
(add and carry normally)

```
  0101
+0011
-----
 1000
```

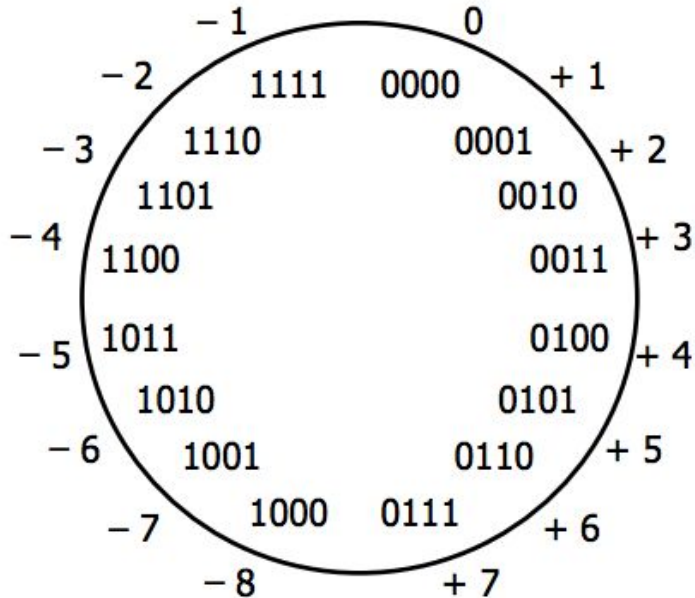
Adding signed ints : (gets
tricky - notice

$4-3 \neq 4+-3$)

```
   0100
+1011
-----
 1111    = 15
```

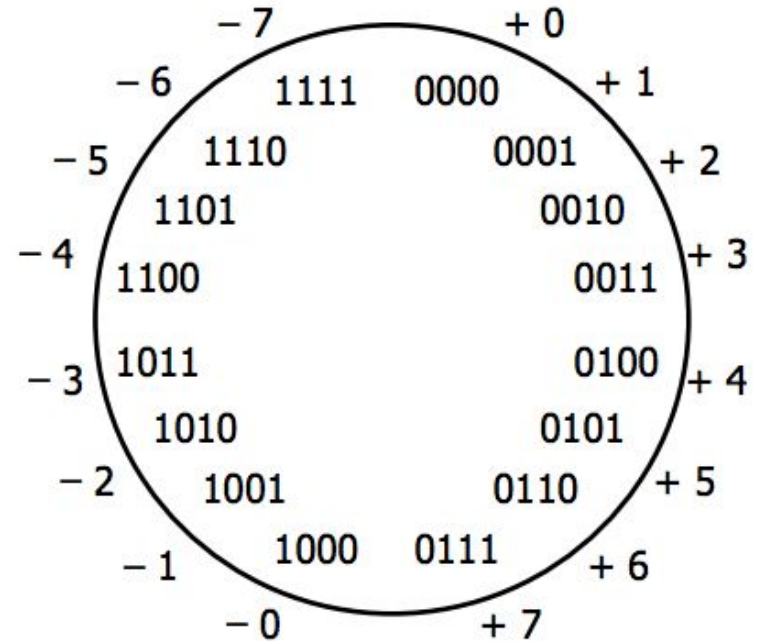


Twos-complement



Imagine the first bit is 'subtract' the value of the digit

Old version - notice the two different representations of '0'



Twos-complement: Benefits

Only 1 representation of 0

Most-significant bit is still the sign

Negate a value

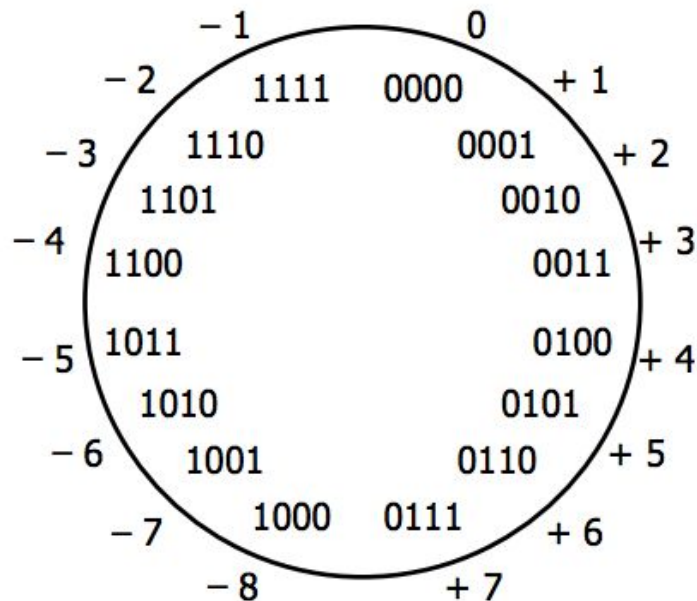
Bitwise complement + 1

$$0101 = 1010 + 1 = 1011$$

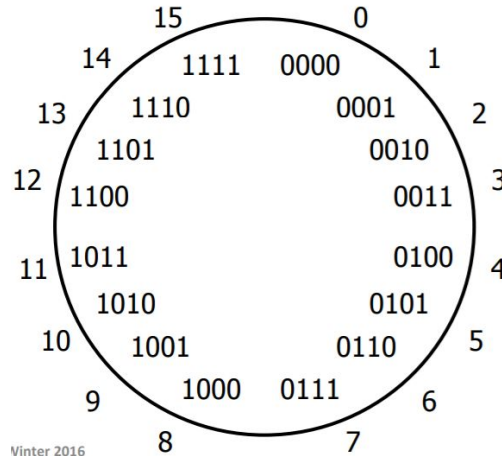
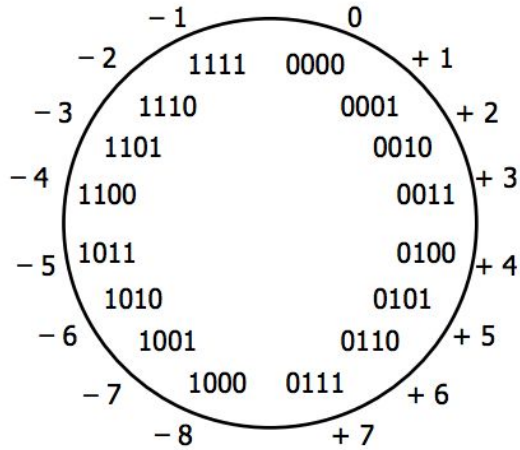
Adding becomes easy again:

$$(4 - 3 = 4 + -3 = 1)$$

$$0100 + 1101 = 0001$$



Twos-complement and unsigned ints



Winter 2016

Get the two-complement number by subtracting 2^w from the unsigned number of the same representation:

$$11 (1011) - 16 = -5 (1011)$$

Use the same algorithm for addition, so hardware implementation is simpler.

What happens if you 'overflow'

Overflow: have numbers too big or small for your number of digits.

(Remember, using 4 bits, unsigned = [0,15] and signed [-8,7])

6+4 = ? (signed)

15+2 = ? (unsigned)

-6 - 6 = ? (signed)

2 - 6 = ? (unsigned)

Notes: You may get a warning for overflow with two-complement numbers, but probably not with unsigned numbers.

```
  0110
+0100
-----
 1010 (-6!)
```

```
  1111
+0010
-----
 0001 (1!)
```

```
  1010
-1010
-----
  0100 (4!)
```

```
  0001
-0110
-----
 1001 (9!)
```

C: 'int' and 'unsigned'

```
int tx, ty;  
unsigned ux, uy;
```

Explicit casting between signed & unsigned:

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

Implicit casting also occurs via assignments and function calls:

```
tx = ux;  
uy = ty;
```

The gcc flag `-Wsign-conversion` produces warnings for implicit casts, but `-Wall` does not!

Casting - doesn't change underlying bits, they just get interpreted differently!

Note: in C, constants are assumed to be signed, unless the 'U' suffix is used: 15U -> 15 unsigned

Float Point Numbers

- Fractional binary numbers work in the same fashion as fractional decimal numbers
 - $1.25 = 1 \cdot 10^0 + 2 \cdot 10^{-1} + 5 \cdot 10^{-2}$
 - $0b1.01 = 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} = 1 + 1/4 = 1.25$
- can have repeating just like decimal
 - $1/10 = 0b0.0001100110011[0011] \dots$
- floating point values only represent numbers that can be written $x \cdot 2^y$
- like scientific notation
 - not $0b0.000101$ but $1.01 \cdot 2^4$
- Floating point standard established
 - 1985, IEEE 754 - before that every system had a different approach

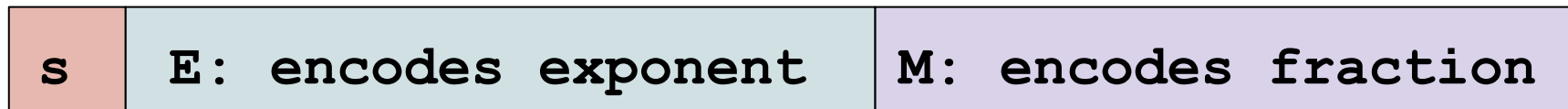
Floating Point Numbers

- Numerical form: $V_{10} = (-1)^s * M * 2^E$
 - Sign bit **s** determines whether number is negative or positive
 - Significand (mantissa) **M** normally a fractional value in range [1.0,2.0)
 - Exponent **E** weights value by a (possibly negative) power of two

s	E: encodes exponent	M: encodes fraction
----------	----------------------------	----------------------------

Floating Point Numbers

- Numerical form: $V_{10} = (-1)^s * M * 2^E$



- For single precision (32 bits), we have s = 1 bit, E = 8 bits, M = 23 bits
- For double precision (64 bits), we have s = 1 bit, E = 11 bits, M = 52 bits
- Since we have a finite number of bits, some values will have to be approximated
- Special values
 - zero: s == 0, E == 0, M == 0
 - $+\infty$, $-\infty$: E == all ones, M == 0
 - NaN (not a number): E = all ones, M != 0
 - special values can pollute numerical computation

Floating Point Numbers

- As with integers, floats suffer from the fixed number of bits available to represent them
 - Can get overflow/underflow, just like ints
- Some “simple fractions” have no exact representation (e.g., 0.2)
- Can also lose precision, unlike ints “Every operation gets a slightly wrong result”
- Mathematically equivalent ways of writing an expression may compute different results
- Violates associativity/distributivity
- Never test floating point values for equality!
- Careful when converting between ints and floats!

Floating Points in C

- C offers two levels of precision
 - float single precision (32-bit)
 - double double precision (64-bit)
- `#include <math.h>` to get INFINITY and NAN constants
- Equality (`==`) comparisons between floating point numbers are tricky
 - often return unexpected results
 - Just avoid them!

Data type conversions

- Conversions between data types:
 - Casting between int, float, and double changes the bit representation.
- int → float
 - May be rounded: overflow not possible
- int → double or float → double
 - Exact conversion (32-bit ints; 52-bit frac + 1-bit sign)
- long int → double
 - Rounded or exact, depending on word size
- double or float → int
 - Truncates fractional part (rounded toward zero)
 - E.g. 1.999 → 1, -1.99 → -1
- “Not defined” when out of range or NaN: generally sets to Tmin

What about Hexadecimal?

Generally use base 10

(10 fingers)

234

$2 \times 100 + 3 \times 10 + 4 \times 1$

$2 \times 10^2 + 3 \times 10^1 + 4 \times 10^0$

Digital systems - base 2

(binary)

$234 = 0b11101010$

**$1 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 +$
 $1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$**

Base 16 - very compact

(hexadecimal)

$234 = 0xEA$

$14 \times 16^1 + 10 \times 16^0$

**Need 16 digits,
so we used [0-9A-F]**

Computer represent things in binary. However, we can capitalize on different representations for compact storage, or for particular needs. One hexadecimal digit takes precisely 4 bits (one nibble) to store. Because 16 corresponds to 2 bytes conversion from binary to hexadecimal is convenient.

Hexadecimal in C

There is no unique type for hexadecimal in C. We use 'unsigned int' or 'unsigned char'.

Remember, `sizeof(int) = 2 or 4 [bytes]`

and `sizeof(char) = 1 [byte]`

An unsigned char can hold values up to 255 or 0xFF (maximum two digit hex value)

```
unsigned char ahexvalue = 0xFE;
uintptr_t mymem = (uintptr_t) malloc (16);
for (int i = 0; i < 16; i++) {
    *((unsigned char*) (mymem+i)) = 0xFE;
}
```

What about `uintptr_t` ?

HW6 - we use 'uintptr_t' as a type to hold a memory address:

`uintptr_t`: Integer type capable of holding a value converted from a void pointer and then be converted back to that type with a value that compares equal to the original pointer.

- Long integer / changes if you move to a different memory model so it is more portable to use these types

What is C++ ?

A big language - much bigger than C

Conveniences in addition to C (new/delete, function overloading, bigger std library)

Namespaces - similar to Java

Extras (casts, exceptions, templates, lambda functions)

Object Oriented - has classes and objects similar to Java

Why C++ ?

- C++ is C-like in
 - User-managed memory
 - Header files
 - Still use pointers
- C++ is Java like in
 - Object Oriented
 - Modern additions to language
- Knowing C++ may help understand both C & Java better