Hexadecimal and number storage
Number systems and BASE

Generally use base 10
(10 fingers)

234
2x100 + 3x10 + 4x1
2x10² + 3x10¹ + 4x10⁰

Digital systems - base 2
(binary)

234 = 0b11101010
1x2⁷ + 1x2⁶ + 1x2⁵ + 0x2⁴ + 1x2³ + 0x2² + 1x2¹ + 0x2⁰

Base 16 - very compact
(hexadecimal)

234 = 0xE0
14x16¹ + 10x16⁰

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.
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: $0 ... 2^W-1$  
  $\leq 2^4-1 \rightarrow 1111 \rightarrow 2^3+2^2+2^1+2^0 \rightarrow 8+4+2+1 \rightarrow 15$
- Signed values: $-2^{W-1} ... 2^{W-1}-1$

Reminder: terminology for binary representations

“Most-significant” / “high-order” bit(s) “Least-significant” / “low-order” bit(s)
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:
(adding and carry normally)

\[
\begin{array}{c}
0101 \\
\text{+0011} \\
\hline
1000
\end{array}
\]

Adding signed ints: (gets tricky - notice
\(4-3 \neq 4+(-3)\))

\[
\begin{array}{c}
0100 \\
\text{+1011} \\
\hline
1111 = 15
\end{array}
\]
Twos-complement

Imagine the first bit is ‘subtract the value of that digit’, so 1111 = (7)-(8), 1010 = (2)-(8)

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

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

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)       12-14 = ? (unsigned)

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

C: ‘int’ and ‘unsigned’

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

Explicit casting between signed & unsigned:

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

Implicit casting also occurs via assignments and function calls:

```c
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! This is NOT taking the absolute value.

Note: C doesn’t dictate the integer representation method, the compiler does. Casting an integer to unsigned will result in different values depending on that choice.

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
  - $1/10 = 0b0.0001100110011[0011]…$
- 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: $V_{10} = (-1)^s \times M \times 2^E$
  - 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: V10 = (−1)^s * M * 2^E

<table>
<thead>
<tr>
<th>s</th>
<th>E: encodes exponent</th>
<th>M: encodes fraction</th>
</tr>
</thead>
</table>

- 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
  - +∞, -∞: 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!

You’ll need to link that at compile time:

```bash
> gcc -lm myprogram.c
```
Data type conversions

- Implicit conversion for math operations ⇒
- 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 1
What about Hexadecimal?

<table>
<thead>
<tr>
<th>Generally use base 10</th>
<th>Digital systems - base 2</th>
<th>Base 16 - very compact</th>
</tr>
</thead>
<tbody>
<tr>
<td>(10 fingers)</td>
<td>(binary)</td>
<td>(hexadecimal)</td>
</tr>
<tr>
<td>234</td>
<td>234 = 0b11101010</td>
<td>234 = 0xEA</td>
</tr>
<tr>
<td>2x100 + 3x10 + 4x1</td>
<td>1x2^7 + 1x2^6 + 1x2^5 + 0x2^4 + 1x2^3 + 0x2^2 + 1x2^1 + 0x2^0</td>
<td>14x16^1 + 10x16^0</td>
</tr>
<tr>
<td>2x10^2 + 3x10^1 + 4x10^0</td>
<td></td>
<td>Need 16 digits, so we used [0-9A-F]</td>
</tr>
</tbody>
</table>

Computers 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. Simultaneously, hex can be easier for humans to read and understand.
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] (2 hex digits)

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

```c
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`?

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