CS 211
Computer Architecture
Fall 2021

David Menendez

Multidimensional Arrays

```
int m[3][3];
```

m is an array of arrays of ints

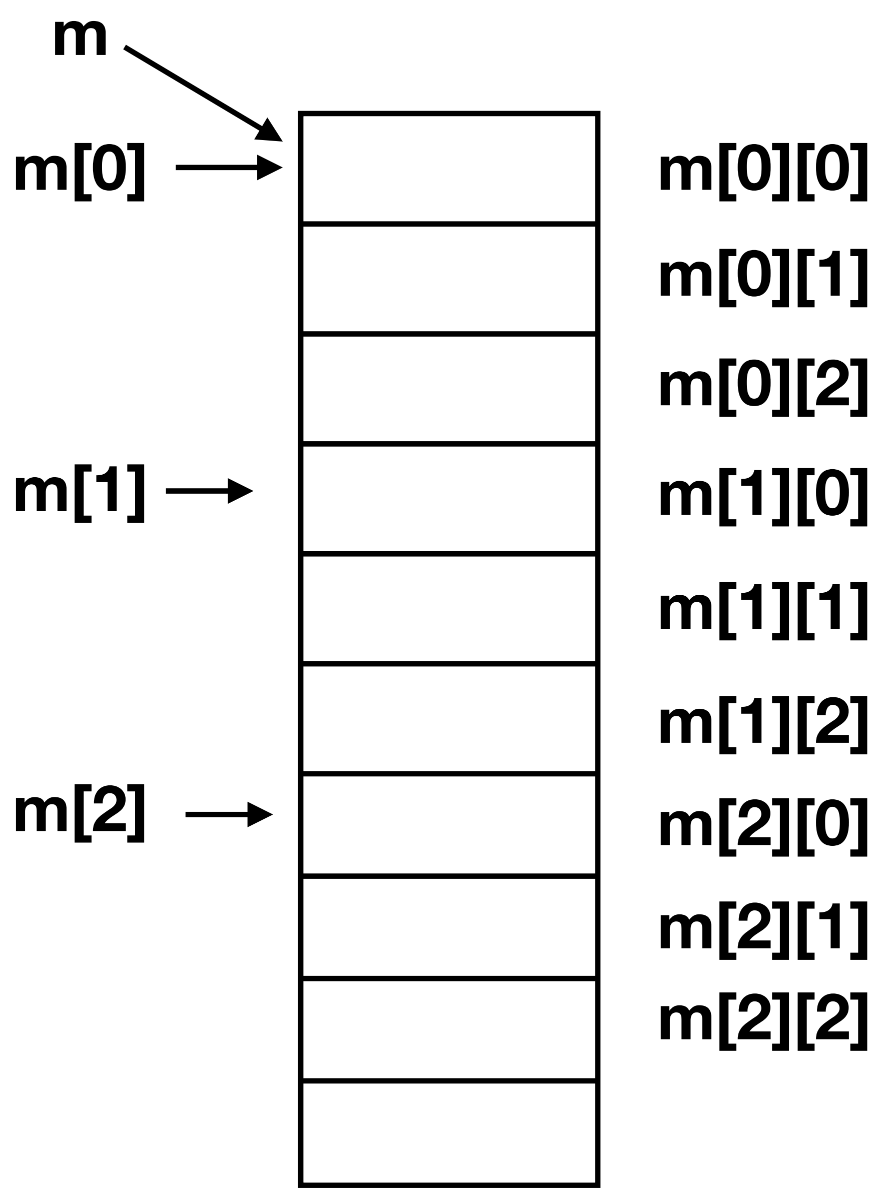
m[0] is an array of ints

m[0][0] is an int

sizeof(m) = ?

sizeof(m[0]) = ?

sizeof(m[0][0]) = ?



Multidimensional Arrays

- To access an element, the compiler needs to know the length of the rows
  - e.g., int m[ROWS][COLS]; the row length is COLS
  - m[row][col] means *(m + row * COLS + col);
  - int v[X_RANGE][Y_RANGE][Z_RANGE];
  - v[x][y][z] means *(v + (x * Y_RANGE + y) * Z_RANGE + z);
- The compiler knows this because of the declaration

Multidimensional Arrays

- How can we pass a multidimensional array to a function?
  - void transpose(double matrix[][3]);
  - void something_else(int voxels[][100][100]);
  - First dimension can be dropped because it isn't needed; others must be given

Double Pointers

- Single pointers and single-dimension arrays can be mostly used interchangeably
- The differences come up in higher dimensions
    - char text[X][Y] is an array of arrays
        - contiguous in memory, rectangular
    - char **argv is a pointer to a pointer
        - No further requirements

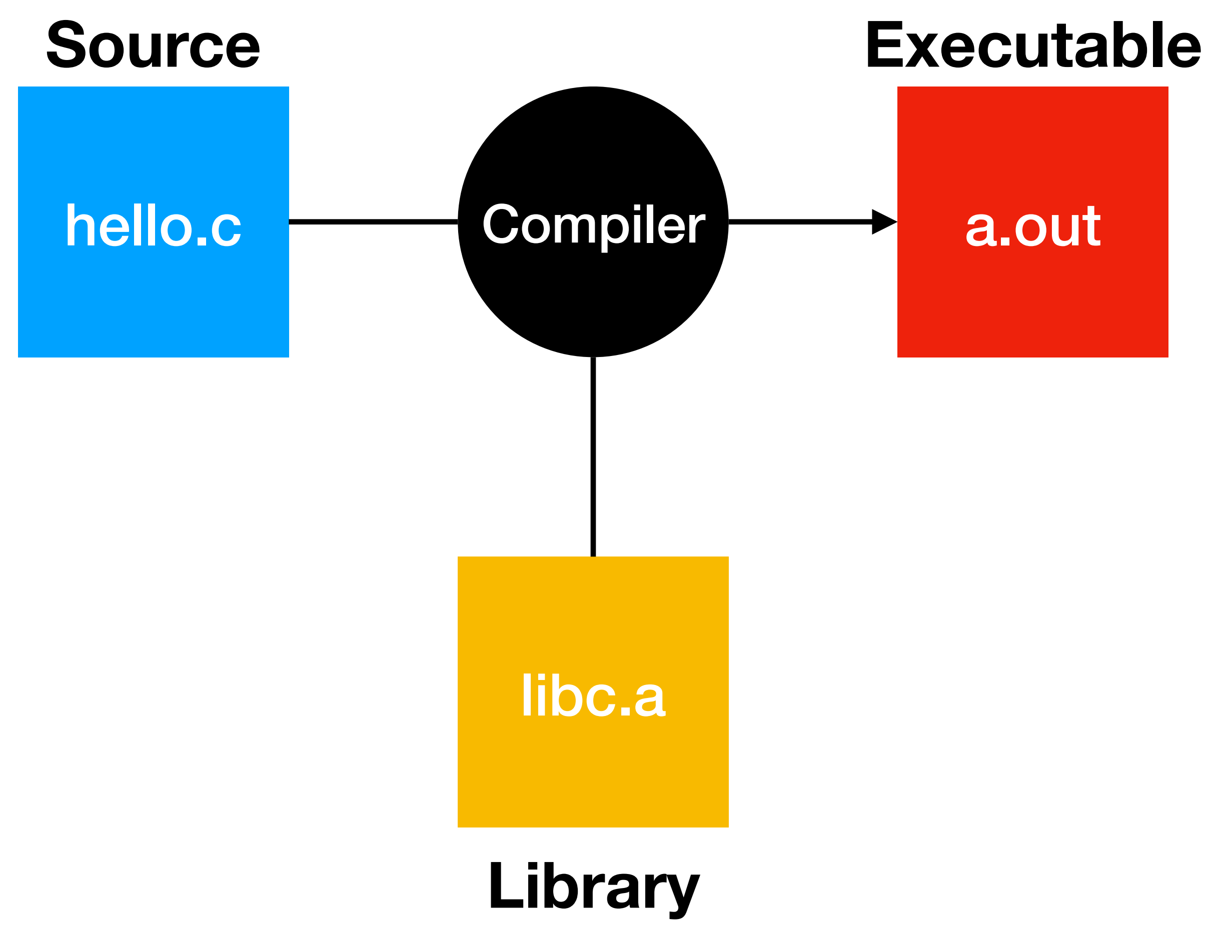
Double Pointers

- Allocating a 2D array with double pointers:

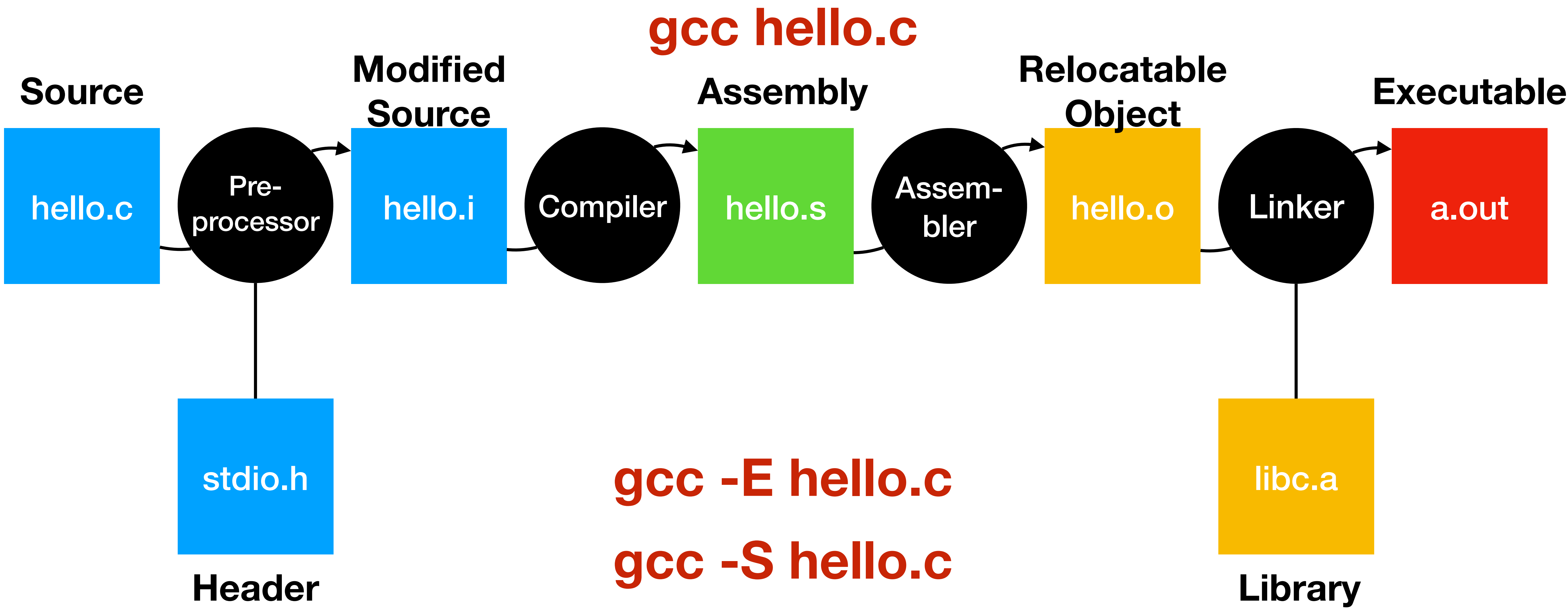
```
double **matrix = malloc(rows * sizeof(double *));

for (i = 0; i < rows; ++i) {

    matrix[i] = malloc(cols * sizeof(double));

}
```

Compilation

gcc hello.c



Compilation

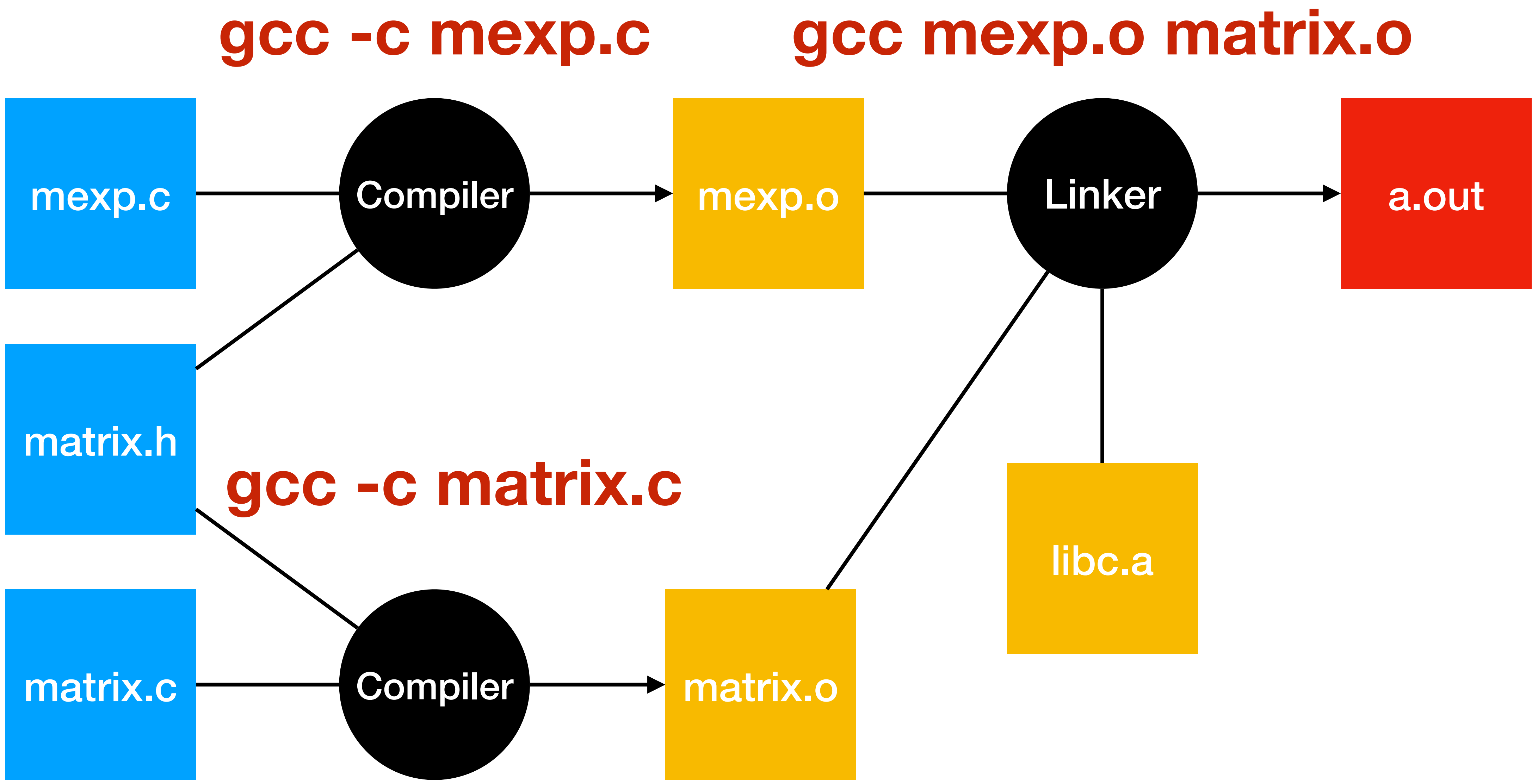


gcc -E hello.c

gcc -S hello.c

gcc -c hello.c

Separate Compilation



Running on Hardware

- What is the interface between software and hardware?
    - ISA: Instruction Set Architecture
- How do we represent data?
    - Data types (integers, floats, pointers, etc.)
    - Instructions themselves are data

What Computers Do

- Computers manipulate data
    - Specifically numbers
    - Most things can be represented using numbers (words, images, sound, etc.)
- Numbers are abstract: we need a representation to manipulate them
    - Many notations used through history
    - Today, decimal notation (base 10) is most common

Base 10: Decimal

- You already know this one
- The right-most (least significant) digit is the 1s place, to its left is the 10s place, to its left is the 100s place, etc.
    - 86042 = 80000 + 6000 + 000 + 40 + 2
    - 86042 = $8 \times 10^4 + 6 \times 10^3 + 0 \times 10^2 + 4 \times 10^1 + 2 \times 10^0$
- We multiply each digit by the *n*th power of 10 (the base), where n is 0 for the right-most digit and increases for each digit to the left
- Side note: this means that initial zeros are irrelevant
    - 100 = 0100 = 00100 = …

Base *n* Notation

- Generalization of decimal notation
- Numbers written as a sequence of digits
  - $d_k d_{k-1} \dots d_2 d_1 d_0$
- Each digit is multiplied by a place value
  - $x = d_k n^k + \dots + d_1 n^1 + d_0 n^0$
- Digit values typically range from 0 to $n-1$

Base 2: Binary

- In binary notation, the only digits are 0 and 1
  - bit ⇐ “binary digit”
- x = $d_k 2^k + \ldots + d_1 2^1 + d_0 2^0$
- Binary is great for computers
  - Easy to represent with switches (on/off)
  - Can be manipulated by digital logic in hardware
- Binary is less good for humans
  - Relatively small numbers require a lot of digits

Base 16: Hexadecimal

- Digits are 0–9 and A–F (representing 10–15)
- $x = d_k 16^k + \dots d_1 16^1 + d_0 16^0$
- More compact than binary
  - $16 = 2^4$, so 1 hex digit is 4 bits
  - Bytes are two hex digits (00–FF)

Hex–Binary Conversion

Hex	Binary	Hex	Binary
0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

Base 8: Octal

- Digits are 0–7
- $x = d_k 8^k + \ldots d_0 8^0$
- More compact than binary
  - One octal digit is 3 bits
- Few things divide into groups of 3 bits these days

Decimal–Binary Conversion

x=13	**1**
x=6	**01**
x=3	**101**
x=1	**1101**

- If *x* is odd, the last bit is 1, otherwise it is 0
- *x*←*x*÷2. If *x* is odd, the next-to-last bit is 1, otherwise 0
- *x*←*x*÷2. If *x* is odd, …

Other Conversions

- In general, repeatedly divide by *n* and take remainder
- Remainder gives digits 0–(*n*–1) starting with d_0
- How would you convert decimal to octal?
- How would you convert binary to base 12?

Base *n* Rationals

- The decimal point extends base 10 integers with additional places smaller than 1 (e.g., 1/10ths place, 1/100ths place)
- Similar "radix points" can be used in base *n*
- Digits after the radix point are multiplied by negative powers of *n*
    - $x = \ldots + d_1 n^1 + d_0 n^0 + d_{-1} n^{-1} + d_{-2} n^{-2} + \ldots$
- 101.11 in binary is 4+0+1+½+¼ = 5.75

Value vs Notation

- Decimal, binary, hexadecimal, etc. are ways of *writing* numbers
- Computers use binary to encode integers
- All arithmetic operates on binary integers
- printf and others convert to decimal etc. when printing
    - Hence %d, %x, %o, etc.
- There are no “hex ints”—the value of any integer can be written in any convenient notation

Data Sizes

- Primitive number types use a fixed number of digits
  - Usually multiples of 8 (8 bits = 1 byte)
- Types vary in C, but char = 1 byte, short int ≥ 2 bytes, long int ≥ 4 bytes

Type	16-bit	32-bit	64-bit
char	1	1	1
short int	2	2	2
int	2	4	4
long int	4	4	8
void *	2	4	8
float	4	4	4
double	8	8	8

Typical, but not universal

Big- and Little-Endian

- Large types consist of multiple bytes
- A 4-byte integer at address *A* will use bytes *A…A*+3
  - But what will be in the specific bytes?
- Consider a large number $AB10CD2F_{16}$
- Big Endian: most significant byte at smallest address:
  - AB 10 CD 2F
- Little Endian: least significant byte at smallest address:
  - 2F CD 10 AB

When Does Endianness Matter?

- The CPU is designed to work with whichever system it has
    - E.g., reads, writes, arithmetic will all work correctly
- But what about sending data to other computers?
    - Need a standard representation for compatibility
    - Most network standards are big-endian
    - Some file formats have byte-order marks

Encoding Data

- How could we encode a playing card?
  - 4 suits, 13 ranks = 52 cards
  - Operation: compare two cards of the same suit
- One byte each for suit and rank
- One byte: 4 bits for rank, 2 bits for suit

Negative Integers

- Base *n* doesn't do negative numbers (if *n* ≥ 0)
- In written notation, we denote negative numbers with –
- You can designate a bit to be a sign bit
  - This is *sign-magnitude* representation
- In 4 bits, 0100 = 4, 1100 = –4, 0110 = 6, 1110 = –6
- But 1000 = –0 — two zero values!
- Inconvenient for arithmetic

1s' Complement

- Idea: to find –x, negate the bits in x
- Still two zeroes
- Inconvenient for arithmetic

000	0	100	–3
001	1	101	–2
010	2	110	–1
011	3	111	–0

2’s Complement

- Idea: Most significant bit is negative
    - In 3 bits, $101_2 = -4 + 0 + 1 = -3$
- To find –x, negate the bits in x and add 1
- Only one zero, but extra minimum value
- +, –, × work without changes
- Used on most computers

000	0	100	–4
001	1	101	–3
010	2	110	–2
011	3	111	–1

Range of 2's Complement

- Given k bits, you can represent 2^k values
- If they are natural numbers, that is $0 - (2^k - 1)$
- What is the range for 2's complement integers?
  - $x = -d_{k-1}2^{k-1} + d_{k-2}2^{k-2} + \ldots d_1 2^1 + d_0 2^0$
  - x is negative iff $d_{k-1} = 1$
- Minimum value is 1000…0, i.e., $-(2^{k-1})$
- Maximum value is 0111…1, i.e., $2^{k-1}-1$

iClicker Quiz

iClicker Quiz 1

- What is the largest unsigned integer in *k* bits?

  A. 2^k

  B. $2^k - 1$

  C. 2^{k-1}

  D. $2^{k-1} - 1$

  E. k

iClicker Quiz 2

- What is the smallest unsigned integer in *k* bits?

  A. 2^k

  B. $2^k - 1$

  C. -2^{k-1}

  D. $-2^k + 1$

  E. 0

iClicker Quiz 3

- What is the largest 2's complement integer in *k* bits?

  A. 2^k

  B. $2^k - 1$

  C. 2^{k-1}

  D. $2^{k-1} - 1$

  E. k

iClicker Quiz 4

- What is the smallest 2's complement integer in *k* bits?

  A. 2^k

  B. $2^k - 1$

  C. -2^{k-1}

  D. $-2^k + 1$

  E. 0

2’s Complement Addition & Subtraction

Addition: same as unsigned binary addition
Subtraction: invert subtrahend and add

```
   -7  1001         -5  1011
+   5  0101      + -2  1110
----------      ----------
   -2  1110         -7 11001
```

4 – 2 = 4 + (–2)

```
     4  0100
+ -2  1110
----------
     2 10010
```

Ignore carries

2’s Complement Overflow

- Addition results can be too large to fit in *k*–1 bits
  - Since we are using regular addition, we overflow into the sign bit

```
    7  0111        -7  1001
+   5  0101     + -7   1001
-----------     -----------
   -4  1100         2 10010
```

  - Detecting overflow:
    - positive + positive ⇒ negative
    - negative + negative ⇒ positive

Does not work in C!

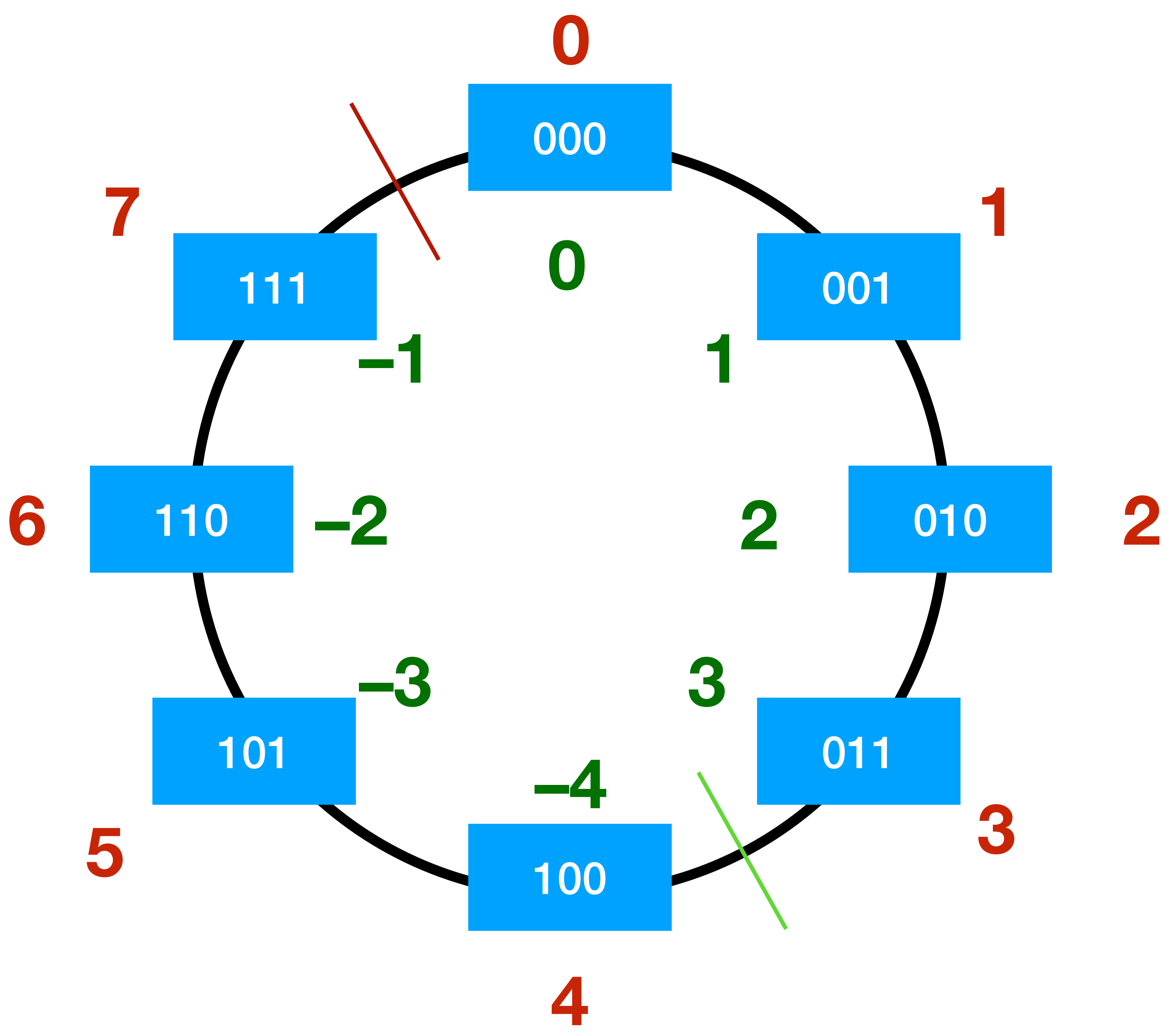
Aside: Modular Arithmetic

- In modular arithmetic, we talk about congruence:
  - $a \equiv b \pmod{m}$
  - This means $\exists d : a = b + md$
- For example: $9 + 9 \equiv 2 \pmod{16}$, because $18 = 2 + 16 \times 1$
- In particular, $-1 = 15 \pmod{16}$, because $-1 = 15 - 16$

Aside: Modular Arithmetic

- Modular arithmetic operators give congruent values in the range 0…*m*–1
    - a + b ≡ c (mod *m*), where 0 ≤ c < *m*
- Modular arithmetic has nearly all the algebraic properties we expect for numbers
    - + and × are commutative and associative
    - (a + b) × c = a × c + a × b
    - a – b = a + (–b)

Modular Numbers



- m+n — start at m, move n spaces clockwise
- m–n — start at m, move n spaces counter-clockwise
- Moving n spaces is the same as moving 8–n spaces the other way

Multiplication

```
   1011  (multiplicand)
×   101  (multiplier)
———————
   1011  (multiplicand × 1)
shift left 0
  0000   (multiplicand × 0)
shift left 1
 1011    (multiplicand × 1)
shift left 2
———————
 110111  (product)
```

Algorithm

1. result ← 0
2. If LSB of multiplier = 1, add multiplicand to result
3. Shift multiplicand left 1 bit (fill LSB with 0)
4. Shift multiplier right 1 bit (fill MSB with 0)
5. If multiplier > 0, go to 2

- We only need to know add and shift to multiply

Divison

```
                 101
              ÷----
15÷3   0011 |1111
       1100  0100  -(divisor << 2)×1
             ----
            10011
       0110  0000  -(divisor << 1)×0
             ----
             0011
       0011  1101  -(divisor << 0)×1
             ----
            10000   remainder
```

(Un)signed Ints in C

- Unsigned values in C
  - type: `unsigned int i = 10;`
  - cast: `i = (unsigned) j;`
- Casting leaves bits unchanged
  - `(unsigned) -1` ⇒ 2^k-1

Sign Extension

Signed Integer	4 bit	8 bit	16 bit
+1	0001	00000001	0000000000000001
–1	1111	11111111	1111111111111111

Bit Shifting

- C operators << and >> shift an integer by some number of places
- n << c returns n shifted c places left
    - $n \ll c = n \times 2^c$
- n >> c returns n shifted c places right
    - $n \gg c = n \div 2^c$
- What about sign?
    - >>, /, and % operate differently for signed and unsigned ints

Base –2

- Idea: even places are positive, odd are negative
- Extremely unbalanced
- No sign extension
- Arithmetic is complicated—even negation
- (Probably) never used anywhere

000	0	100	4
001	1	101	5
010	–2	110	2
011	–1	111	3

Floating Point

Scientific Notation

- Another way of writing numbers
  - $1{,}250{,}000 = 1.25 \times 10^6$
  - Can be more compact
  - Distinguish magnitude from precision
- Can be used with base *n* for any *n*
  - $110001.1_2 \approx 1.1_2 \times 2^5$; $0.00101_2 = 1.01_2 \times 2^{-3}$
- How might we represent this in computers?

Fixed-Point Numbers

- Choose an exponent e and represent $m \times b^e$ as an integer
    - For example, \$10.50 = \$1050 $\times 10^{-2}$
- Use integer operations to add and subtract
- To multiply, use integer multiplication and then multiply by b^e
    - $(45 \times 10^{-2})(1000 \times 10^{-2}) = (45 \times 1000 \times 10^{-2}) \times 10^{-2}$
      $= 450 \times 10^{-2}$

Floating-Point Numbers

- Idea: Represent numbers as $m \times b^e$
- Represent *m* and *e* as (fixed-width) integers
    - Base *b* is fixed (usually 2, sometimes 10)
- Allows for a much wider range of values
- Variable precision
    - Values closer to 0 are more precise — why?

IEEE 754 Floating-Point

- The most commonly-used representation for floating-point
- Designed by experts in numerics
    - Has lots of features that surprise non-experts
- Defines 3 standard precisions for binary FP:
    - single (32 bit), double (64 bit), extended (80 bit)
- Some processors also support half (16 bit) or quad (128 bit)

Normal Values

- All IEEE FP numbers have three parts
    - sign *s*, exponent *e*, significand *m*
    - s is 1 bit; e and m depend on the precision
- Conceptually: $(-1)^s \times m \times 2^{e-\text{bias}(E)}$, where $1 \leq m < 2$
    - $\text{bias}(E) = 2^{E-1}-1$, where E is the number of bits in e
    - Yet another way to represent signed numbers
    - $e \neq 0$ and $e \neq 2^E-1$(e.g., all ones)

Other Values

- Zero and “denormal” values — less than any normal value
  - Conceptually: $(-1)^{s} \times m \times 2^{1-\text{bias}(E)}$, where $0 \leq m < 1$
  - ±0 if $m = 0$ and $e = 0$
  - Denormal if $m \neq 0$ and $e = 0$
- Non-finite values
  - $\pm\infty$ if $m = 0$ and $e = 2^{E}-1$ (all ones)
  - NaN if $m \neq 0$ and $e = 2^{E}-1$

Conversion to FP

- Single precision: E = 8, M = 23 (mantissa bits)
- Rewrite 5.625 in binary: 101.101_2
- Rewrite in scientific notation: $1.01101_2 \times 2^2$
- Solve e – bias(E) = 2
  - In single precision, bias(E) = 127, so e = 129 = 10000001_2
- Drop the integer part of m (we can deduce it using e) and extend to M bits: $01101000000000000000000_2$
- Thus: 0 100000001 01101000000000000000000

Question 1

If 1011 is a 4-bit, 2’s complement integer, what is its value?

A. 11

B. 0

C. –3

D. –4

E. –5

Question 2

IEEE double-precision floating point has how many bits?

A. 1

B. 16

C. 23

D. 32

E. 64

Question 3

What is 1.101×2^1 in decimal?

A. 1.625

B. 3.25

C. 6.5

D. 13

E. –3.75

Question 4

What was the hardest part of PA1?

A. Memory management

B. Input/output

C. Using the command-line tools

D. Accessing the iLab

E. Time management