CS 110
Computer Architecture

Lecture 2: *Introduction to C I*

Instructors:
Sören Schwertfeger & Chundong Wang

School of Information Science and Technology SIST

ShanghaiTech University

Slides based on UC Berkley's CS61C

Agenda

- Everything is a Number
- Compile vs. Interpret
- Pointers
- Pointers & Arrays

Agenda

- Everything is a Number
- Compile vs. Interpret
- Pointers
- Pointers & Arrays

BIG IDEA: Bits can represent anything!!

- Characters?
  - 26 letters $\Rightarrow$ 5 bits ($2^5 = 32$)
  - upper/lower case + punctuation
    $\Rightarrow$ 7 bits (in 8) ("ASCII")
  - standard code to cover all the world's languages $\Rightarrow$ 8,16, 32 bits ("Unicode")
    www.unicode.com

- Logical values?
  - 0 $\rightarrow$ False, 1 $\rightarrow$ True
- colors ? Ex: ***Red (00)*** ***Green (01)*** ***Blue (11)***
- locations / addresses? commands?
- MEMORIZE: N bits $\Leftrightarrow$ at most 2^N things

Key Concepts

- Inside computers, everything is a number
- But numbers usually stored with a fixed size
  - 8-bit bytes, 16-bit half words, 32-bit words, 64-bit double words, …
- Integer and floating-point operations can lead to results too big/small to store within their representations: *overflow*

Number Representation

- Value of i-th digit is $d \times Base^{i}$ where i starts at 0 and increases from right to left:
- $123_{10} = 1_{10} \times 10_{10}^{2} + 2_{10} \times 10_{10}^{1} + 3_{10} \times 10_{10}^{0}$

  $= 1x100_{10} + 2x10_{10} + 3x1_{10}$

  $= 100_{10} + 20_{10} + 3_{10}$

  $= 123_{10}$
- Binary (Base 2), Hexadecimal (Base 16), Decimal (Base 10) different ways to represent an integer
  - We use 1_{two}, 5_{ten}, 10_{hex} to be clearer (vs. 1_2, 4_8, 5_{10}, 10_{16})

Number Representation

- Hexadecimal digits:
  0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
- $FFF_{hex} = 15_{ten} \times 16_{ten}^{2} + 15_{ten} \times 16_{ten}^{1} + 15_{ten} \times 16_{ten}^{0}$
  $= 3840_{ten} + 240_{ten} + 15_{ten}$
  $= 4095_{ten}$
- $1111\ 1111\ 1111_{two} = FFF_{hex} = 4095_{ten}$
- May put blanks every group of binary, octal, or hexadecimal digits to make it easier to parse, like commas in decimal

Signed and Unsigned Integers

- C, C++, and Java have *signed integers*, e.g., 7, -255:

```
int x, y, z;
```

- C, C++ also have *unsigned integers*, e.g. for addresses
- 32-bit word can represent 2^{32} binary numbers
- Unsigned integers in 32 bit word represent 0 to 2^{32}-1 (4,294,967,295) (4 Gig)

Unsigned Integers

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = 2_{ten}$
... ...
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = 2{,}147{,}483{,}645_{ten}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = 2{,}147{,}483{,}646_{ten}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = 2{,}147{,}483{,}647_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 2{,}147{,}483{,}648_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 2{,}147{,}483{,}649_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = 2{,}147{,}483{,}650_{ten}$
... ...
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = 4{,}294{,}967{,}293_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = 4{,}294{,}967{,}294_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = 4{,}294{,}967{,}295_{ten}$

Signed Integers and Two's-Complement Representation

- Signed integers in C; want ½ numbers <0, want ½ numbers >0, and want one 0
- *Two's complement* treats 0 as positive, so 32-bit word represents 2^{32} integers from -2^{31} (–2,147,483,648) to $2^{31}-1$ (2,147,483,647)
  - Note: one negative number with no positive version
  - Book lists some other options, all of which are worse
  - Every computer uses two's complement today
- *Most-significant bit* (leftmost) is the *sign bit*, since 0 means positive (including 0), 1 means negative
  - Bit 31 is most significant, bit 0 is least significant

Two's-Complement Integers

Sign Bit

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = 2_{ten}$
… …
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = 2{,}147{,}483{,}645_{ten}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = 2{,}147{,}483{,}646_{ten}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = 2{,}147{,}483{,}647_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -2{,}147{,}483{,}648_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -2{,}147{,}483{,}647_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = -2{,}147{,}483{,}646_{ten}$
… …
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = -3_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -2_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -1_{ten}$

Ways to Make Two's Complement

- For N-bit word, complement to 2_{ten}^{N}
  - For 4 bit number $3_{ten}=0011_{two}$, two's complement

    (i.e. -3_{ten}) would be

    $16_{ten}-3_{ten}=13_{ten}$ or $10000_{two} - 0011_{two} = 1101_{two}$
- Here is an easier way:
  - Invert all bits and add 1
  - Computers actually do it like this, too

3_{ten} 0011_{two}

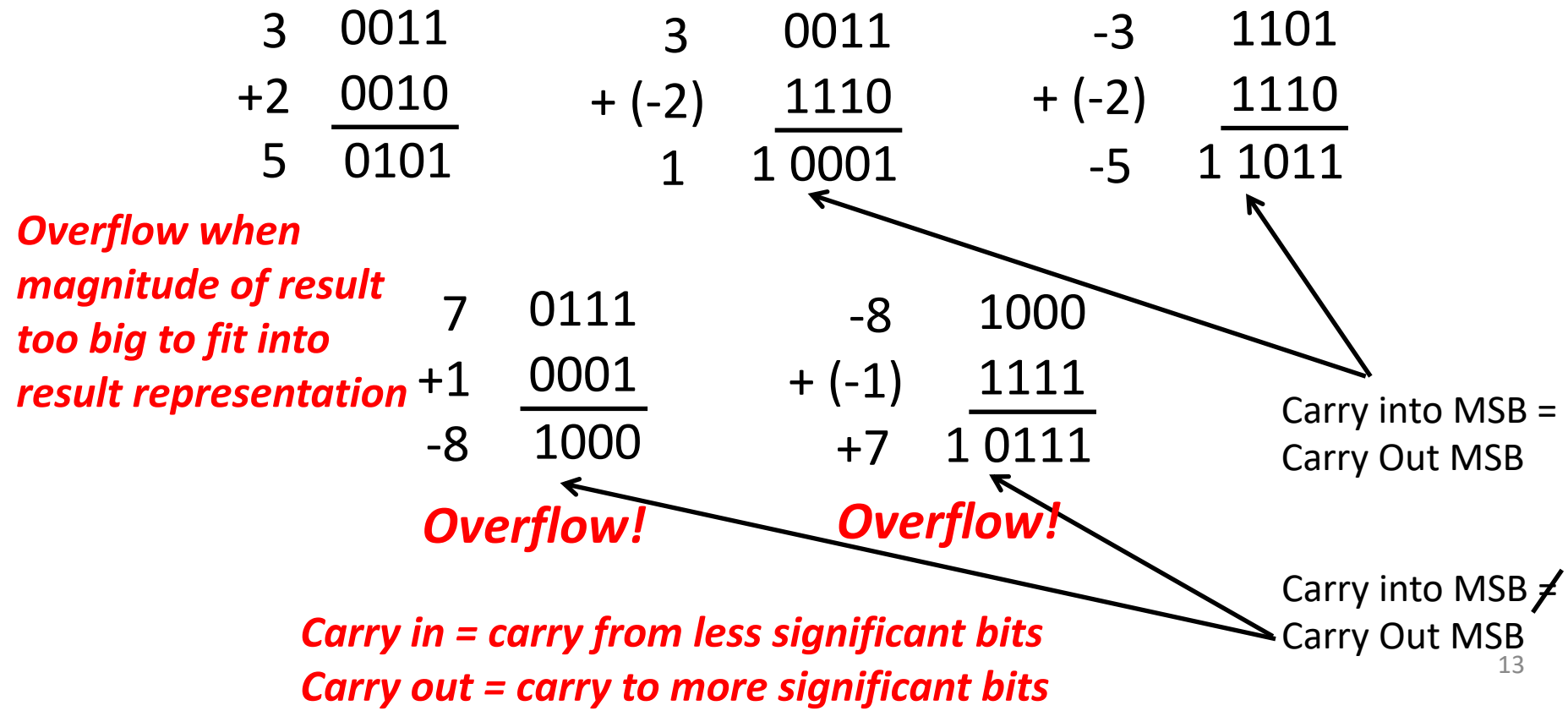
Bitwise complement 1100_{two}

$+ \quad 1_{two}$

-3_{ten} 1101_{two}

Two's-Complement Examples

- Assume for simplicity 4 bit width, -8 to +7 represented



Suppose we had a 5-bit word. What integers can be represented in two's complement?

A -32 to +31

B 0 to +31

C -16 to +15

D -15 to +16

Suppose we had a 5-bit word. What integers can be represented in two's complement?

A -32 to +31

B 0 to +31

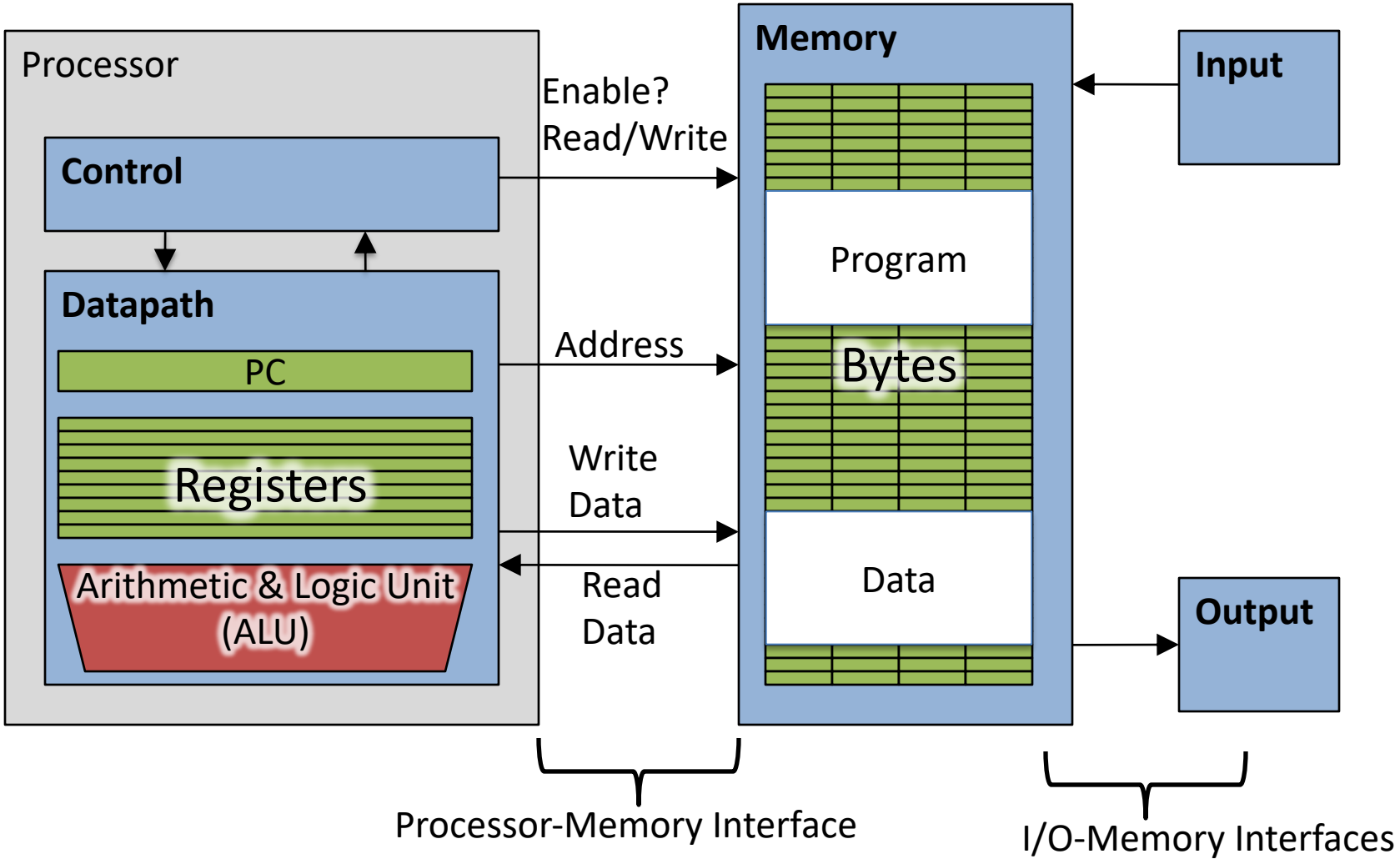
C -16 to +15

D -15 to +16

Agenda

- Everything is a Number
- Compile vs. Interpret
- Pointers
- And in Conclusion, …

Components of a Computer



Great Idea: Levels of Representation/Interpretation

High Level Language Program (e.g., C)

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

We are here!

Compiler

Assembly Language Program (e.g., RISC-V)

```
lw    t0, 0(s2)
lw    t1, 4(s2)
sw    t1, 0(s2)
sw    t0, 4(s2)
```

Assembler

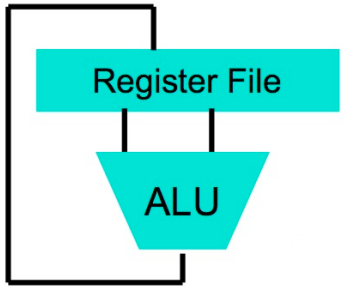
Machine Language Program (RISC-V)

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

Anything can be represented as a *number*, i.e., data or instructions

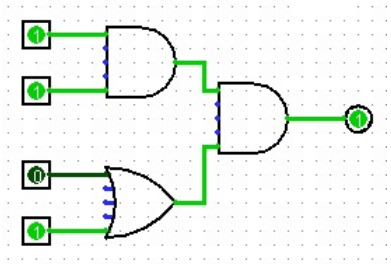
Machine Interpretation

Hardware Architecture Description (e.g., block diagrams)

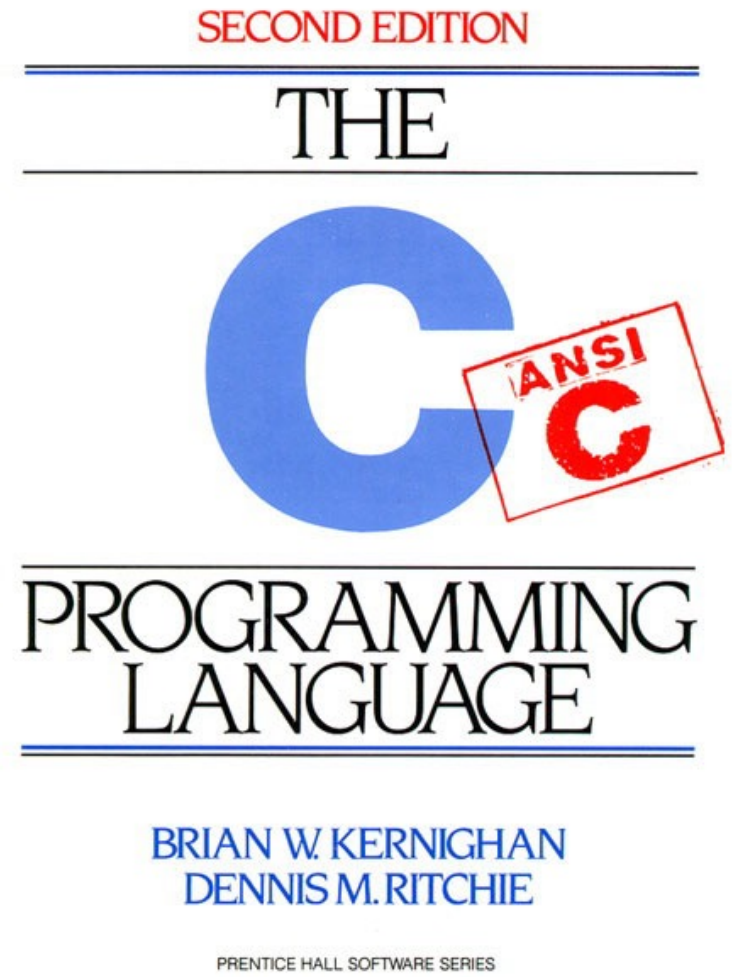


Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)



Introduction to C
“The Universal Assembly Language”



Intro to C

- *C is not a "very high-level" language, nor a "big" one, and is not specialized to any particular area of application. But its absence of restrictions and its generality make it more convenient and effective for many tasks than supposedly more powerful languages.*

  – Kernighan and Ritchie

- Enabled first operating system not written in assembly language: *UNIX* - A portable OS!

Intro to C

- Why C?: *we can write programs that allow us to exploit underlying features of the architecture – memory management, special instructions, parallelism*
- C and derivatives (C++/Obj-C/C#) still one of the most popular application programming languages after >40 years!

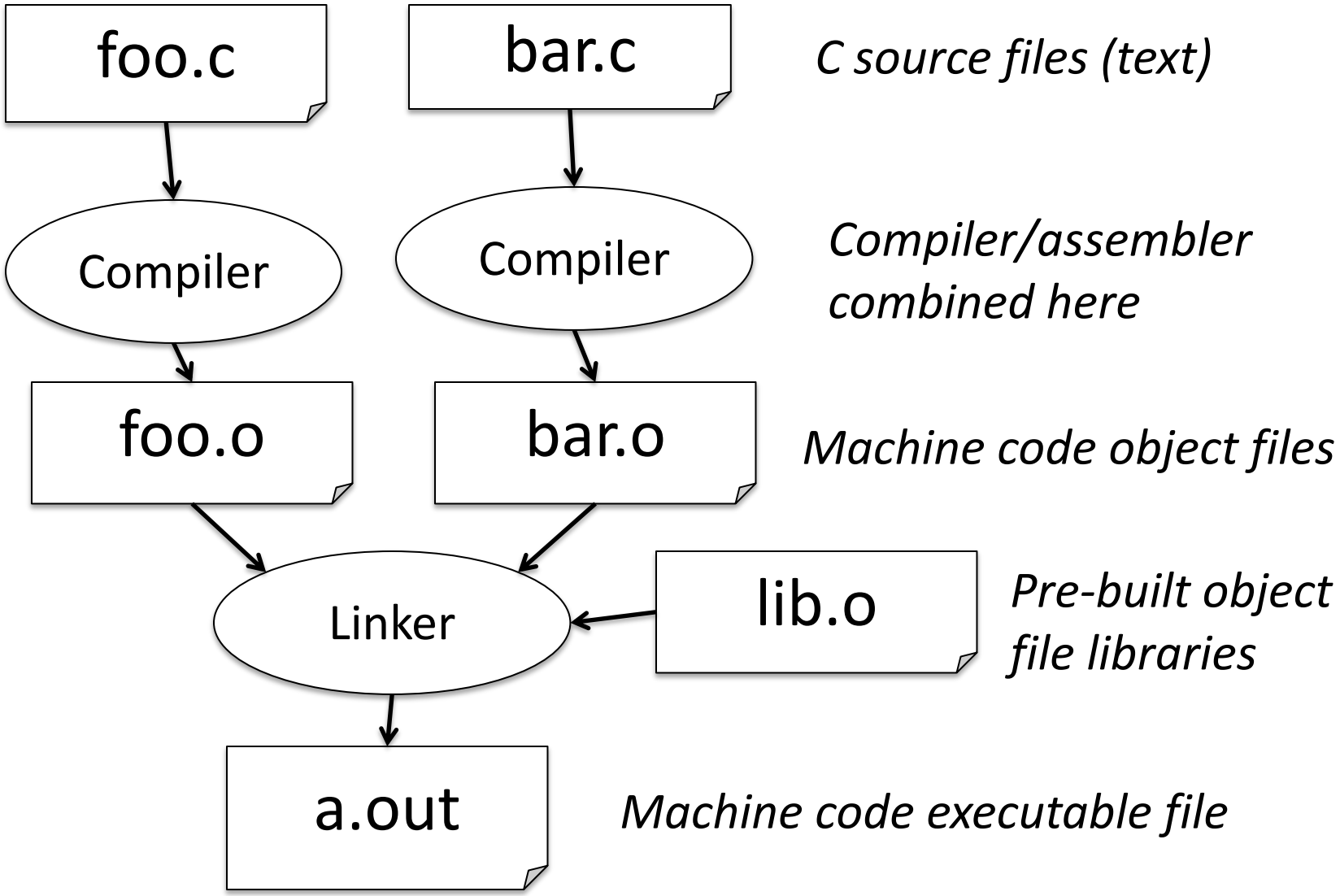
Disclaimer

- You will not learn how to fully code in C in these lectures! You'll still need your C reference for this course
  - K&R is a must-have
    - Check online for more sources
- Key C concepts: Pointers, Arrays, Implications for Memory management
- We will use ANSI C89 – original "old school" C
  - Because it is closest to Assembly

Compilation: Overview

- C *compilers* map C programs into architecture-specific machine code (string of 1s and 0s)
  - Unlike *Java*, which converts to architecture-independent *bytecode*
  - Unlike *Python* environments, which *interpret* the code
  - These differ mainly in exactly when your program is converted to low-level machine instructions ("levels of interpretation")
  - For C, generally a two part process of compiling .c files to .o files, then linking the .o files into executables;
  - Assembling is also done (but is hidden, i.e., done automatically, by default); we'll talk about that later

C Compilation Simplified Overview (more later in course)



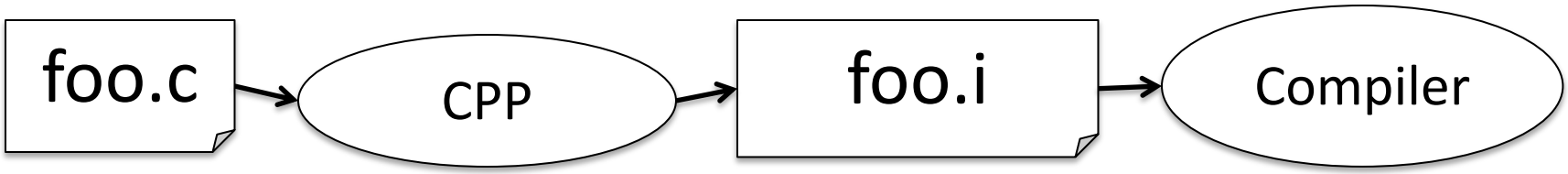
Compilation: Advantages

- Excellent run-time performance: generally much faster than Scheme or Java for comparable code (because it optimizes for a given architecture)
- Reasonable compilation time: enhancements in compilation procedure (Makefiles) allow only modified files to be recompiled

Compilation: Disadvantages

- Compiled files, including the executable, are architecture-specific, depending on processor type (e.g., ARM vs. RISC-V) and the operating system (e.g., Windows vs. Linux)
- Executable must be rebuilt on each new system
  - I.e., "porting your code" to a new architecture
- "Change → Compile → Run [repeat]" iteration cycle can be slow during development
  - but Make tool only rebuilds changed pieces, and can do compiles in parallel (linker is sequential though -> Amdahl's Law)

C Pre-Processor (CPP)



- C source files first pass through macro processor, CPP, before compiler sees code
- CPP replaces comments with a single space
- CPP commands begin with “#”
- #include “file.h” /* Inserts file.h into output */
- #include <stdio.h> /* Looks for file in standard location */
- #define M_PI (3.14159) /* Define constant */
- #if/#endif /* Conditional inclusion of text */
- Use -save-temps option to gcc to see result of preprocessing
- Full documentation at: **http://gcc.gnu.org/onlinedocs/cpp/**

CPP Macro
Which one is the correct way?

```
// Magnitude (Length) of Vector (x, y)
1) #define mag(x, y) = sqrt( x*x + y*y );
2) #define mag(x, y) = sqrt( x*x + y*y )
3) #define mag(x, y) = (sqrt( x*x + y*y ))
4) #define mag(x, y) sqrt( x*x + y*y );
5) #define mag(x, y) sqrt( x*x + y*y )
6) #define mag(x, y) (sqrt( x*x + y*y ))
7) #define mag(x, y) = sqrt( (x)*(x) + (y)*(y) )
8) #define mag(x, y) = sqrt( (x*x) + (y*y) );
9) #define mag(x, y) sqrt( (x*x) + (y*y) )
10)#define mag(x, y) (sqrt( (x*x) + (y*y) );)
11)#define mag((x), (y)) (sqrt( (x*x) + (y*y) ))
```

CPP Macro

- Correct answer: **NONE**
- Most correct solution:

```
#define MAG(x, y) (sqrt( (x)*(x) + (y)*(y)))
```

- Rules:
  - Convention: macros are CAPITALIZED
  - Put parenthesis around arguments – if missing:
  - `#define MAG(x, y) (sqrt( (x*x) + (y*y)))`
  - `MAG( a+2, 1-b) =>`
    `sqrt( (a+2*a+2) + (1-b*1-b) =>`
    `sqrt( (3*a+2) + (1-2*b))`

CPP Macro

- More Pitfalls:
  - Put the whole macro body in parentheses:
    - `#define ADD(a, b) (a) + (b)`
    - `int result = 3 * ADD( 2, 3); // is 15!? =>`
    - `int result = 3 * 2 + 3; // is 9`
  - => Convention: put parenthesis EVERYWHERE!
  - Never put a semicolon after the macro:
    - `#define MAG(x, y) (sqrt( (x)*(x) + (y)*(y)));`
    - May work for:
    - `double len = MAG(a+2, 1-b);`
    - But not, for example, here:
    - `printf("Magnitude: %f ", MAG(a, b));`

CPP Macro II

- Most Correct version:
- `#define MAG(x, y) (sqrt( (x)*(x) + (y)*(y)))`
- `int val = 3;`
- `double len = MAG(++val, 4);`
- `Printf(" val: %d  len^2: %f \n", val, len*len);`

A: val: 3 len^2: 25 B: val: 4 len^2: 25 C: val: 5 len^2: 25

D: val: 3 len^2: 32 E: val: 4 len^2: 32 F: val: 5 len^2: 32

G: val: 3 len^2: 36 H: val: 4 len^2: 36 I: val: 5 len^2: 36

J: val: 3 len^2: 41 K: val: 4 len^2: 41 L: val: 5 len^2: 41

M: val: 6 len^2: 25 N: val: 6 len^2: 32 O: val: 6 len^2: 36

P: val: 6 len^2: 41

CPP Macro II

- Most Correct version:
- `#define MAG(x, y) (sqrt( (x)*(x) + (y)*(y)))`
- `int val = 3;`
- `double len = MAG(++val, 4);`
- `Printf(" val: %d  len^2: %f \n", val, len*len);`
- **Answer: I: val: 5 len^2: 36:**

```
double len = (sqrt((++val)*(++val) + (4)*(4)));
double len = (sqrt((4)*(5) + (4)*(4)));
```

```
test.c:14:19: warning: multiple unsequenced modifications to 'val'
      [-Wunsequenced]
       double len = MAG(++val, 4);
                        ^~
test.c:8:27: note: expanded from macro 'MAG'
#define MAG(x, y) (sqrt( (x)*(x) + (y)*(y)))
                          ^   ~
1 warning generated.
```

CPP Macro II

- Avoid using macros whenever possible
- NO or very tiny speedup.
- Instead use C functions – e.g. inline function:

```
double mag(double x, double y);
double inline mag(double x, double y)
{  return sqrt( x*x + y*y ); }
```

- Read more…
- https://chunminchang.gitbooks.io/cplusplus-learning-note/content/Appendix/preprocessor_macros_vs_inline_functions.html

Typed Variables in C

```
int   variable1   = 2;
float variable2   = 1.618;
char  variable3   = 'A';
```

- Must declare the type of data a variable will hold
  - Types can't change

Type	Description	Examples
int	integer numbers, including negatives	0, 78, -1400
unsigned int	integer numbers (no negatives)	0, 46, 900
long	larger signed integer	-6,000,000,000
char	single text character or symbol	'a', 'D', '?'
float	floating point decimal numbers	0.0, 1.618, -1.4
double	greater precision/big FP number	10E100

Integers: Python vs. Java vs. C

Language	sizeof(int)
Python	>=32 bits (plain ints), infinite (long ints)
Java	32 bits
C	Depends on computer; 16 or 32 or 64

- C: `int` should be integer type that target processor works with most efficiently
- Only guarantee: sizeof(`long long`) ≥ sizeof(`long`) ≥ sizeof(`int`) ≥ sizeof(`short`)
  - Also, `short` >= 16 bits, `long` >= 32 bits
  - All could be 64 bits

Consts and Enums in C

- Constant is assigned a typed value once in the declaration; value can't change during entire execution of program

```
const float golden_ratio = 1.618;
const int days_in_week = 7;
```

- You can have a constant version of any of the standard C variable types
- Enums: a group of related integer constants. Ex:

```
enum cardsuit {CLUBS,DIAMONDS,HEARTS,SPADES};
enum color {RED, GREEN, BLUE};
```

Compare "`#define PI 3.14`" and "`const float pi=3.14`" – which is true?

A: Constants "PI" and "pi" have same type

B: Can assign to "PI" but not "pi"

C: Code runs at same speed using "PI" or "pi"

D: "pi" takes more memory space than "PI"

E: Both behave the same in all situations

C Syntax: Variable Declarations

- *All* variable declarations must appear before they are used (e.g., at the beginning of the block)
- A variable may be initialized in its declaration; if not, it holds garbage!
- Examples of declarations:
  - Correct:
```
{
    int a = 0, b = 10;
    ...
}
```
  - Incorrect:
```
for (int i = 0; i < 10; i++)
```

Newer C standards are more flexible about this…

C Syntax: True or False

- What evaluates to FALSE in C?
  - 0 (integer)
  - NULL (a special kind of *pointer*: more on this later)
  - *No explicit Boolean type*
- What evaluates to TRUE in C?
  - Anything that isn't false is true
  - Same idea as in Python: only 0s or empty sequences are false, anything else is true!

C operators

- arithmetic: +, -, *, /, %
- assignment: =
- augmented assignment: +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=
- bitwise logic: ~, &, |, ^
- bitwise shifts: <<, >>
- boolean logic: !, &&, ||
- equality testing: ==, !=
- subexpression grouping: ( )
- order relations: <, <=, >, >=
- increment and decrement: ++ and --
- member selection: ., ->
- conditional evaluation: ? :

ADMIN

HW 1

- New firewall still prevents autolab server to send emails … once its fixed HW1 will be published…

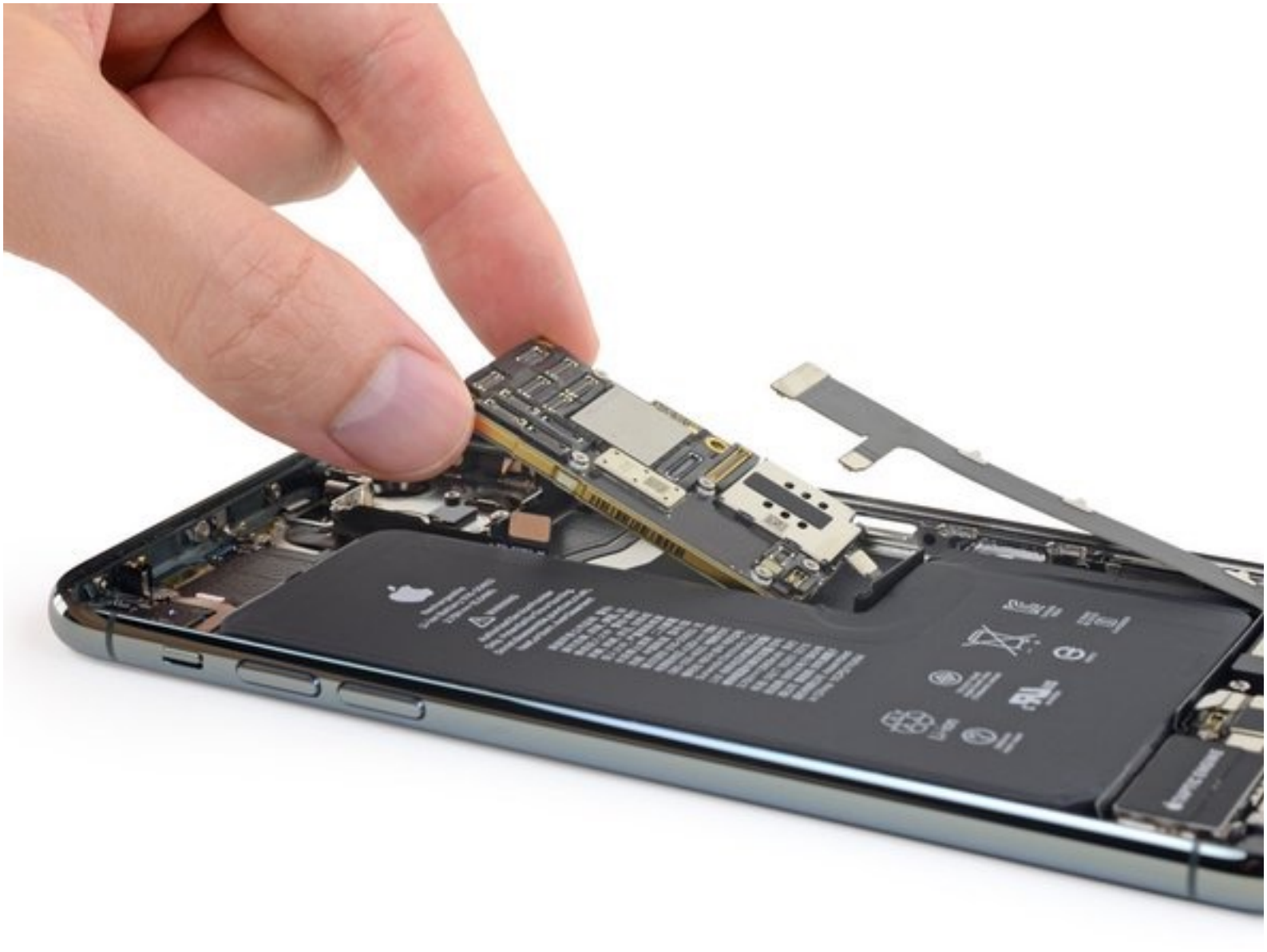
- Discussion time on Monday needs to be pushed back 30 minutes: starts 9pm!

- Discussions start next week

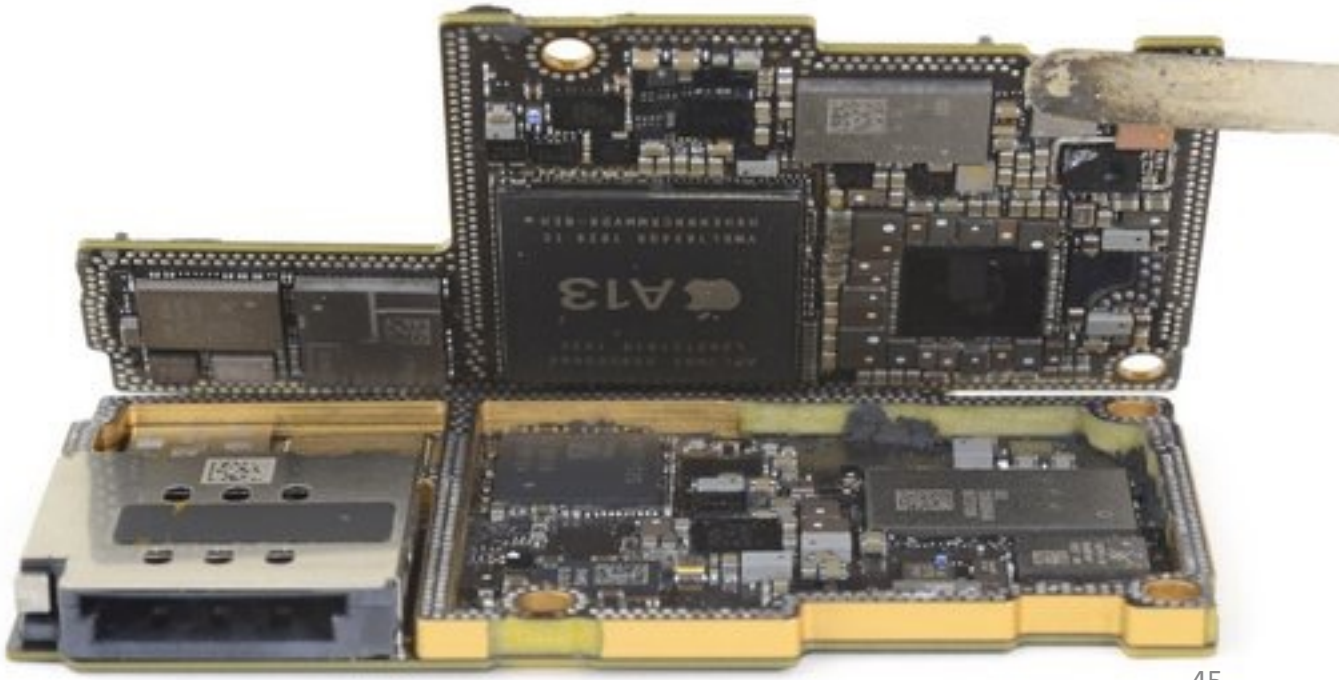
iPhone 11 Pro Max Teardown

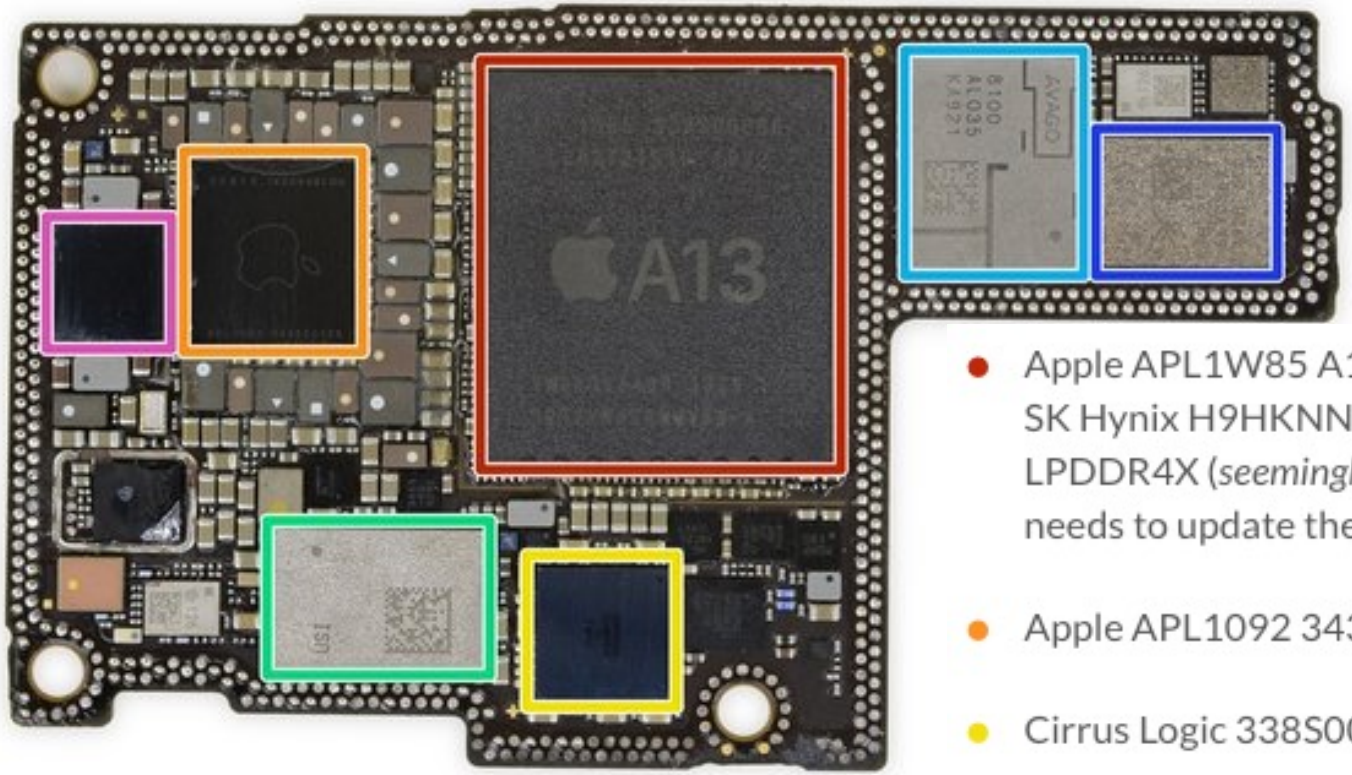
ifixit.com

Get logic board out
dual layer design



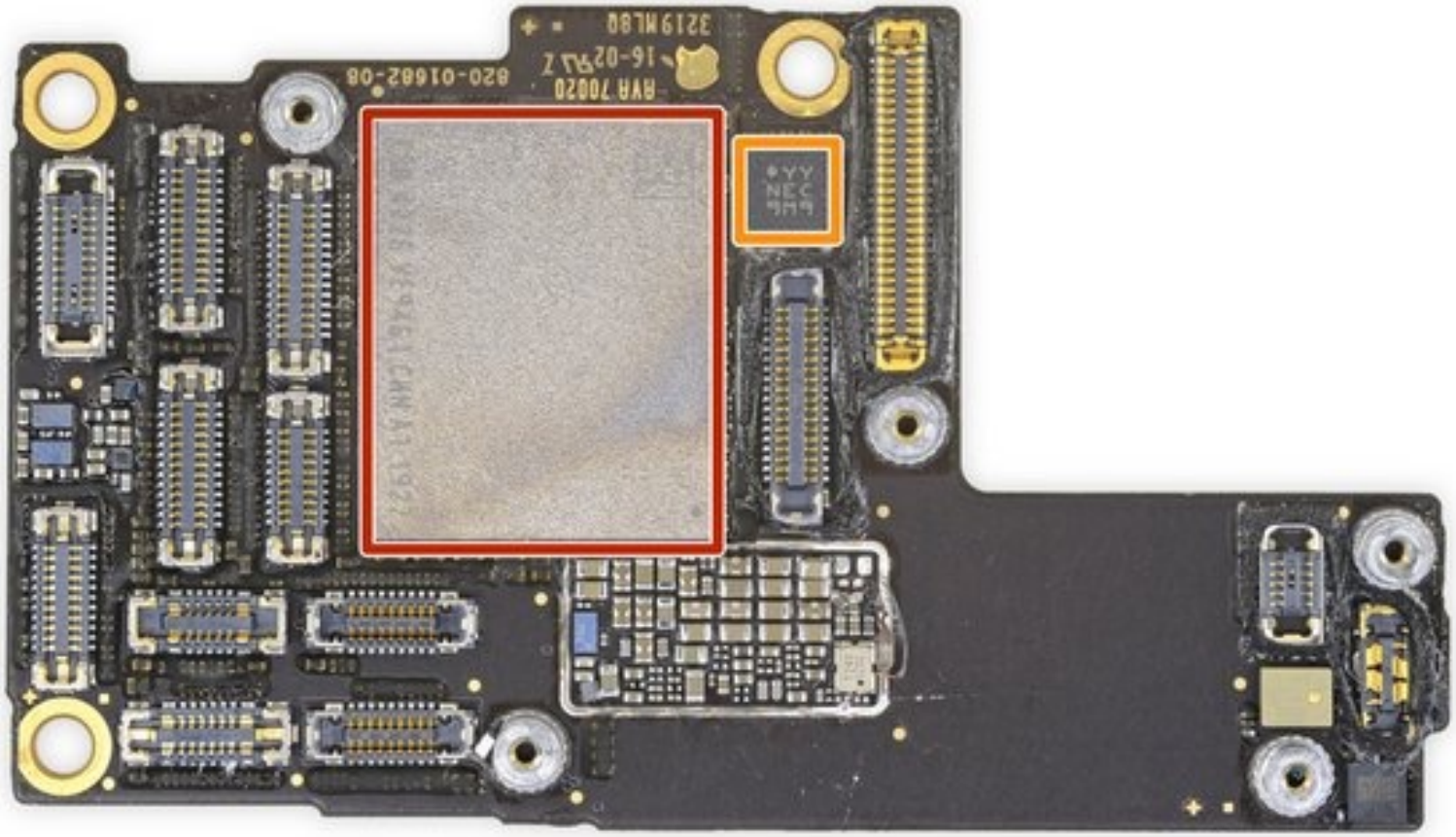
- Apple 64bit System on a chip (SoC); A13:
  - Hexa core (2 high performance (up to 2.66 GHz), 4 low power)
  - Apple designed GPU
  - Motion Processor; Image Processor; Neural Engine
  - 4 GB LPDDR4X (memory)
  - L1 cache: 128 KB instruction, 128 KB data (fast cores)
  - L2 cache: 8 MB; (fast cores; 4MB slow cores)
  - L3 cache : yes, 16MB, shared with other cores (e.g. GPU)

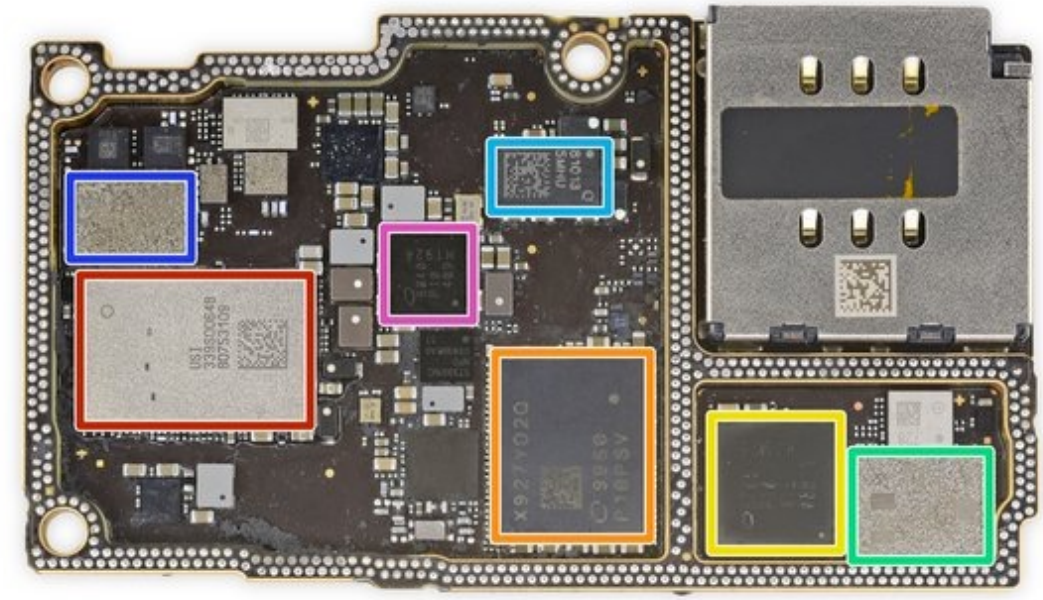


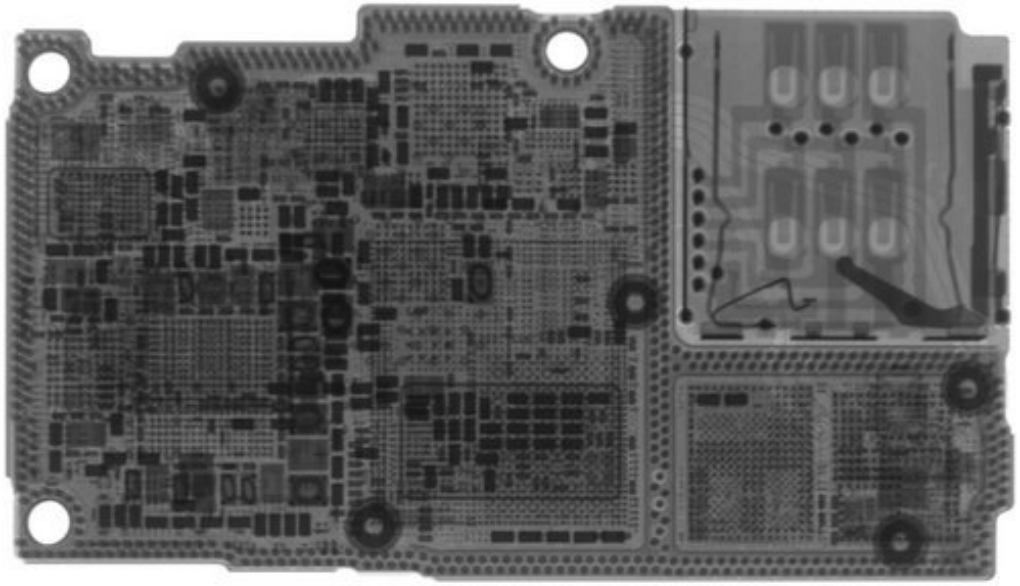


- Apple APL1W85 A13 Bionic SoC layered over SK Hynix H9HKNNNCRMMVDR-NEH LPDDR4X (*seemingly* 4 GB, but SK Hynix needs to update their decoder)
- Apple APL1092 343S00355 PMIC
- Cirrus Logic 338S00509 audio codec
- Unmarked USI module—**teardown update**: it turns out that this *is* where Apple's new U1 ultra-wideband chip is hiding. Read all about it about it in our blog post.
- Avago 8100 Mid/High band PAMiD
- Skyworks 78221-17 low-band PAMiD
- STMicrolectronics STB601A0N power management IC

- Toshiba TSB 4226VE9461CHNA1 1927 64 GB flash storage
- YY NEC 9M9 (likely accel/gyro)







RF board

- Apple/USI 339S00648 WiFi/Bluetooth SoC
- Intel X927YD2Q (likely XMM7660) modem
- Intel 5765 P10 A15 08B13 H1925 transceiver
- Skyworks 78223-17 PAM
- 81013 - Qorvo Envelope Tracking
- Skyworks 13797-19 DRx
- Intel 6840 P10 409 H1924 baseband PMIC


IFIXIT