Goals for today

- Build process
  - How do source files become an executable?
  - What does the linker do?
- Diagnosing build errors
- Modules and libraries
  - What makes for good design?

- Admin
  - Keep eye on piazza!
  - Strategies for assign3

Where are we going?

Processor and memory architecture

Peripherals: GPIO, timers, UART

Assembly language and machine code

From C to assembly language

Function calls and stack frames

Serial communication and strings

Modules and libraries: Building and linking

Memory management: Memory map & heap

gpio
timer
uart
strings
printf
malloc
keyboard
fb
gl
console
shell

Good modules

Meaningful decomposition of the system into parts (modules)

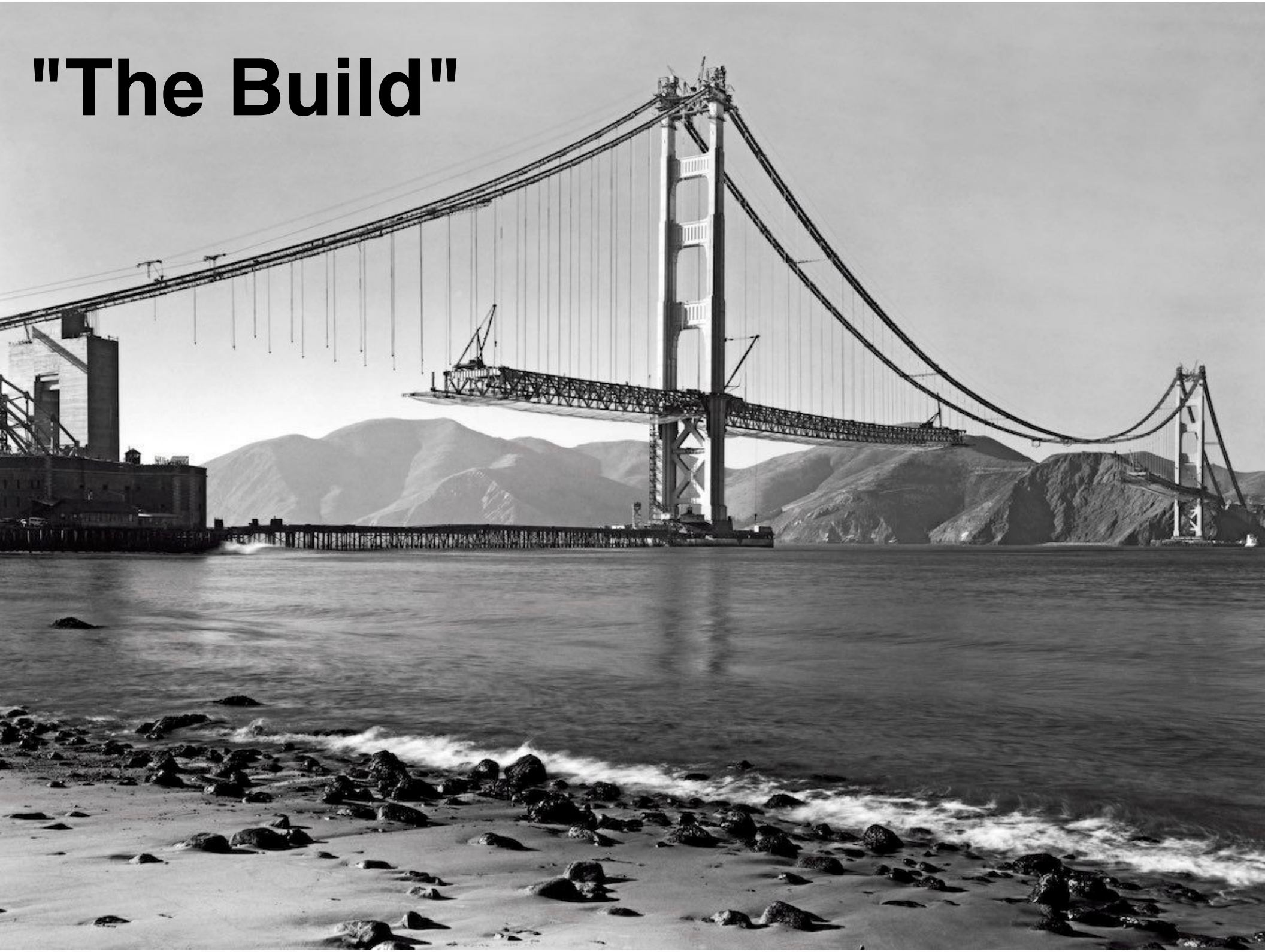
Interfaces and implementations

- Provide easy-to-use interface to users
- Hide obscure details of implementation

Tested independently with unit tests

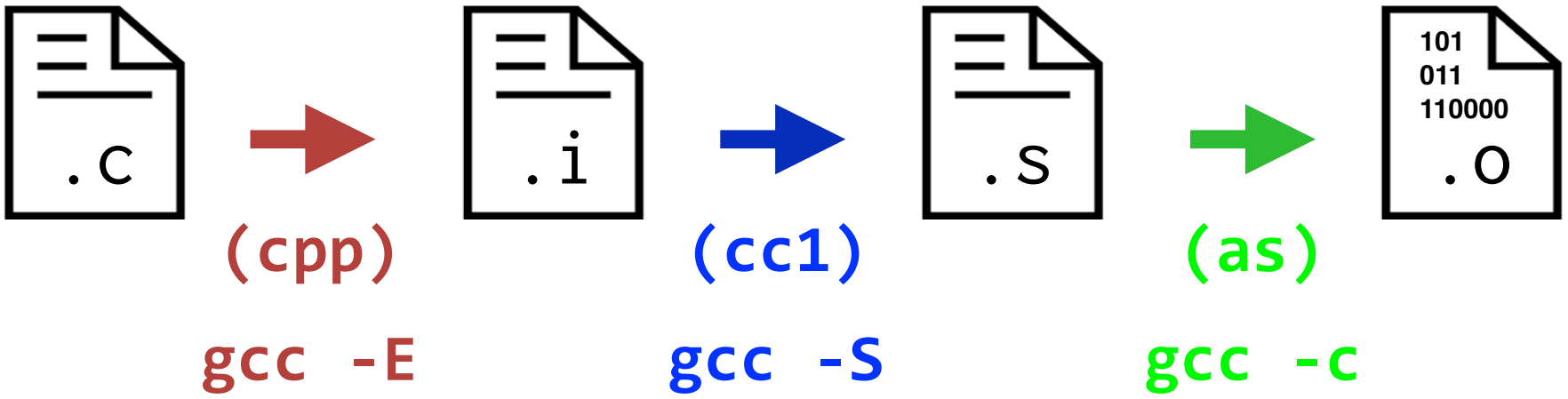
Obi-wan, how should I design my modules?

"The Build"



```
APPLICATION = clock
MY_MODULES = timer.o gpio.o

CFLAGS = -Og -g -Wall -std=c99 -ffreestanding
LDFLAGS = -nostdlib -T memmap
LDLIBS =

all : $(APPLICATION).bin

%.bin: %.elf
	arm-none-eabi-objcopy $< -O binary $@

%.elf: %.o $(MY_MODULES) start.o cstart.o
	arm-none-eabi-gcc $(LDFLAGS) $^ $(LDLIBS) -o $@

%.o: %.c
	arm-none-eabi-gcc $(CFLAGS) -c $< -o $@

%.o: %.s
	arm-none-eabi-as $< -o $@
```

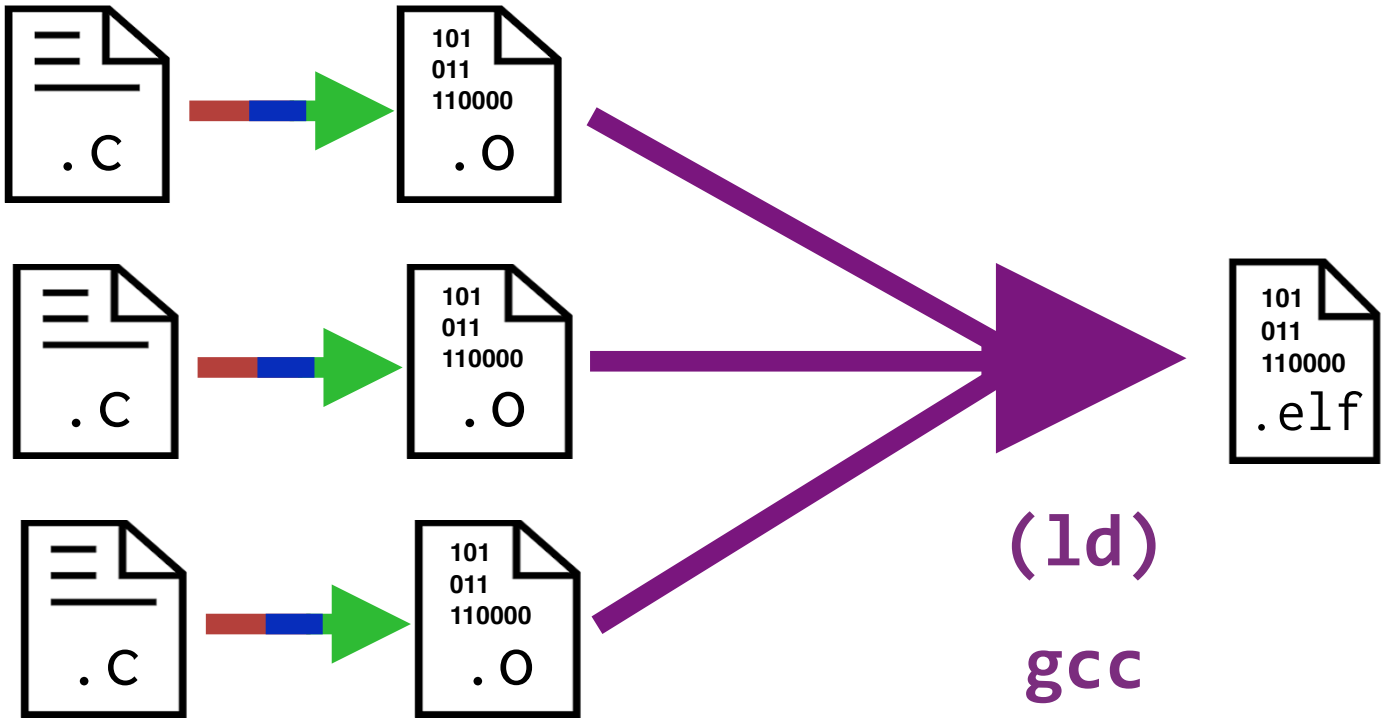
Compiling a single module



want to see intermediate files?

`gcc --save-temps`

Linking modules into executable




```
gcc clock.o timer.o gpio.o start.o cstart.o
```

What does the linker do?

Combines one or more object files into executable

Resolve inter-module references

Consolidate code, data sections across all modules

Arrange sections in output file in proper order

Symbols

Single global namespace

- A name can have only one definition
- Need conventions to avoid clashes
  - e.g. `gpio_init` versus `timer_init`

Qualifier dictates visibility, linking

- **`static`** private to module, not visible outside, no linking needed
- **`extern`** public, exported for use by other modules

Linker only concerned with extern symbols

Symbol resolution

Rule: Every symbol referenced must be defined once and exactly once

Possible errors:

Multiply-defined: two definitions for same symbol

Undefined: needed symbol never defined

Linker resolution process

Set D (symbols that have been defined)

Set U (symbols that have been referenced, but not yet defined)

Linker processes .o files in order from left to right

For each .o, updates **Set U** and **Set D** and goes on to next

until all processed

Link successful: Set U empty and set D contains no duplicates!

Libraries

An archive .a is just a collection of modules (.o files)

The linker scans the library to find definitions for symbols in U (the undefined set). When a symbol is found, the entire module and all its functions are linked in.

If adding that module results in more undefined symbols; linker searches library for definition of these symbols and links those modules; repeats until no more definitions of undefined symbols are found, linker moves on to next library

Diagnosing build errors

What tool detects/reports… ?

mismatched type

function call with wrong arguments

forgot #include

forgot to link with library

use variable uninitialized

#include "wrongname.h"

two functions with same name

Building with libpi

```
APPLICATION = main
MY_MODULES = printf.o strings.o

CS107E = ../cs107e.github.io/cs107e
CFLAGS = -I$(CS107E)/include -g -Wall -Og -std=c99 -ffreestanding
LDFLAGS = -nostdlib -T memmap -L$(CS107E)/lib
LDLIBS = -lc_pi -lpi -lgcc

all : $(APPLICATION).bin $(MY_MODULES)

%.bin: %.elf
	arm-none-eabi-objcopy $< -O binary $@

%.elf: %.o $(MY_MODULES) start.o cstart.o
	arm-none-eabi-gcc $(LDFLAGS) $^ $(LDLIBS) -o $@
...
```

Combining multiple modules (`.o`) into a single executable (`.elf`)

`memmap`

```
SECTIONS
{
    .text 0x8000 :  { start.o(.text*)
                      *(.text*) }
}
```

Why must `start.o` go first?

What is the significance of `0x8000`?

```
% arm-none-eabi-nm -n clock.elf
00008000 T _start
0000800c t hang
00008010 T square
0000801c T blink
00008070 T main
0000808c T timer_init
00008090 T timer_get_ticks
00008098 T timer_delay_us
000080a4 T timer_delay_ms
000080c0 T timer_delay
000080e0 T gpio_init
000080e4 T gpio_set_function
000080e8 T gpio_get_function
000080f0 T gpio_set_input
000080f4 T gpio_set_output
000080f8 T gpio_write
000080fc T gpio_read
00008104 T _cstart
```

`size` reports the size of the segments:

```
% arm-none-eabi-size clock.elf
   text    data     bss     dec     hex filename
    340       0       0     340     154 clock.elf

% arm-none-eabi-size *.o
   text    data     bss     dec     hex filename
    124       0       0     124      7c clock.o
     80       0       0      80      50 cstart.o
     36       0       0      36      24 gpio.o
     16       0       0      16      10 start.o
     84       0       0      84      54 timer.o
```

Note sum of sizes for .o's equals size of main.elf

Relocation

Relocation

In the .o:

- each symbol assigned a temporary address (offset within module)
- access to other symbols in same module is PC-relative

In the executable:

- all modules consolidated
- first module is laid down at fixed address (0x8000)
- base for each module is offset by size of previous modules
- final address of each symbol =

  module_base + symbol_offset_within_module

```
// start.s

.globl _start
_start:
    mov sp, #0x8000000
    mov fp, #0
    bl _cstart
hang: b hang
```

```
// Disassembly of start.o

0000000 <_start>:
   0:   mov sp, #0x80000000
   4:   mov fp, #0
   8:   bl  0 <_cstart>

0000000c <hang>:
   c:   b   c <hang>
```

Note: the address of **_cstart** is 0. Why?

It does know the address of **hang**. What's different between the two?

```
// Disassembly of clock.elf

<_start>:
    8000:   mov     sp, 0x8000000
    8004:   bl      80c8 <_cstart>

<hang>:
    8008:   b       8008 <hang>
```

Address of **`_cstart`** now known — **`80c8`**
Who figured that out?

How is data handled?

Factors to consider:

- extern vs static
- read-only vs read-write
- initialized vs uninitialized

```
                                // uninitialized
int gNum;
static int sgNum;

                                // initialized
int iNum = 1;
static int siNum = 2;

                                // const
const int cNum = 3;
static const int scNum = 4;
```

```
% arm-none-eabi-nm -S data.o
00000000 00000034 T binky
00000000 00000004 R cNum
00000004 00000004 C gNum
00000004 00000004 D iNum
00000034 0000001c T main
00000000 00000004 b sgNum
00000000 00000004 d siNum
```

The global uninitialized gNum is common (C).

The static const scNum was optimized out!

Guide to symbols (nm)

`T/t` - text (code)

`D/d` - read-write data

`R/r` - read-only data (**`const`**)

`B/b` - bss (*Block Started by Symbol*)

`C` - common (instead of B)

lowercase letter means static

```
SECTIONS
{
    .text 0x8000 :  { start.o(.text*)
                      *(.text*) }
    .data :         { *(.data*) }
    .rodata :       { *(.rodata*) }

    __bss_start__ = .;
    .bss :          { *(.bss*)
                      *(COMMON) }
    __bss_end__ = ALIGN(8);
}
```

Sections

Instructions go in `.text`

Data goes in `.data`

const data (read-only) goes in `.rodata`

Uninitialized data goes in `.bss`