Admin

Assign 2

In the bag, y'all are killing it 💪

Assign 3

`printf` code jam Sun 2-6pm

Wrangle those C-strings, yeah!

Today: Modules, Linking

Modules and libraries

- What makes for good design?

Build process

- How do source files become an executable?
- What does the linker do?
- Understanding and diagnosing build errors

Start sequence

- How does a program begin executing?

Road map

Instruction set architecture

Peripherals: GPIO, timers, UART

Assembly language and machine code

From C to assembly language

Function calls and stack frames

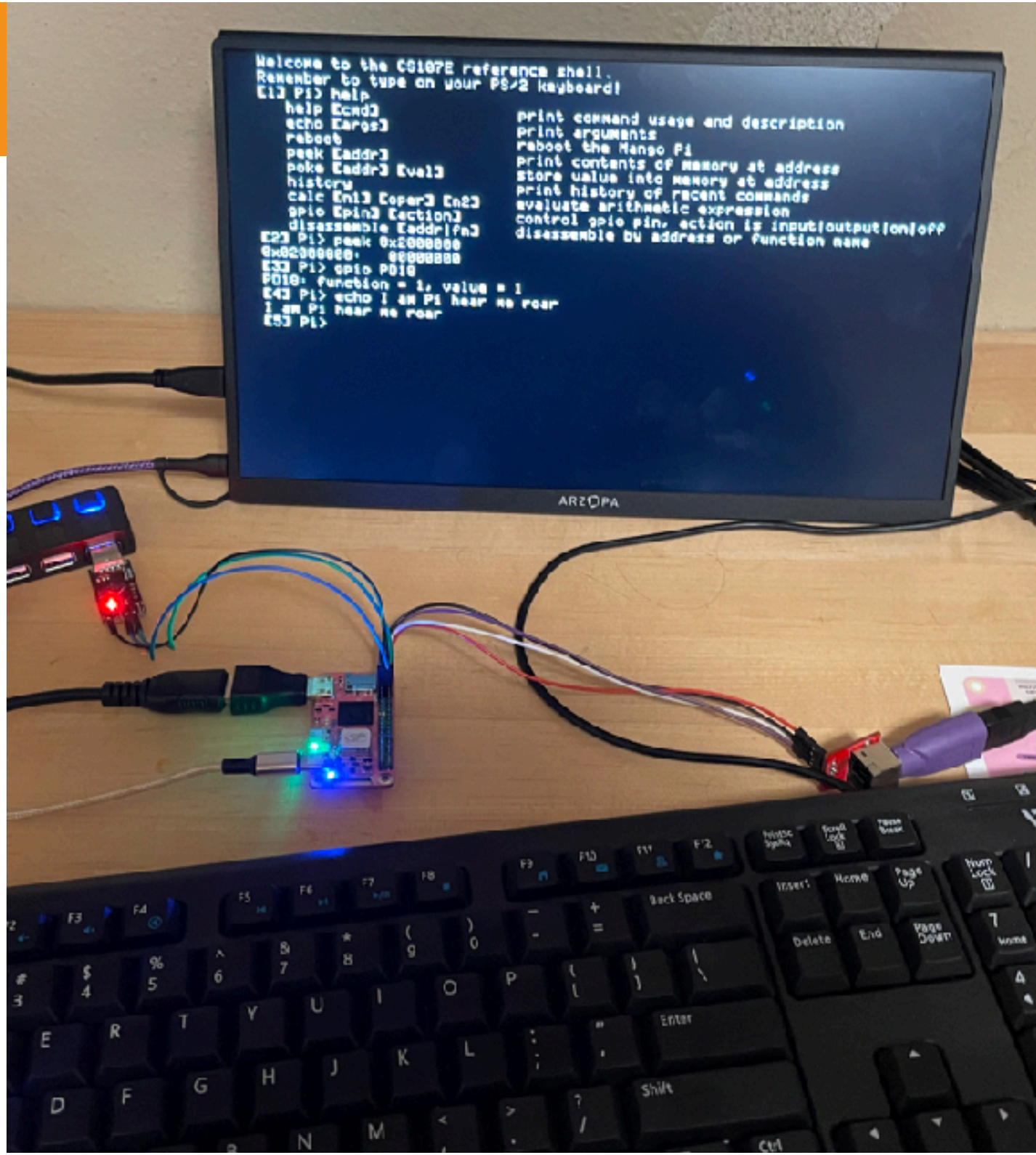
Serial communication and strings

➜ **Modules and libraries: Building and linking**

Memory map, stack & heap

libmango

- gpio ✓
- timer ✓
- uart ✓
- strings
- printf
- malloc
- keyboard
- shell
- fb
- gl
- console



Good software design

Decompose a system into smaller parts (modules)

- **Interface**: what the module does
- **Implementation**: how it does it

A good interface:

- Easy-to-understand abstraction that simplifies client use
- Clear spec, portable, can swap implementations

Testing contract, module unit tests

Designing good interface boundaries is the "art" of software engineering

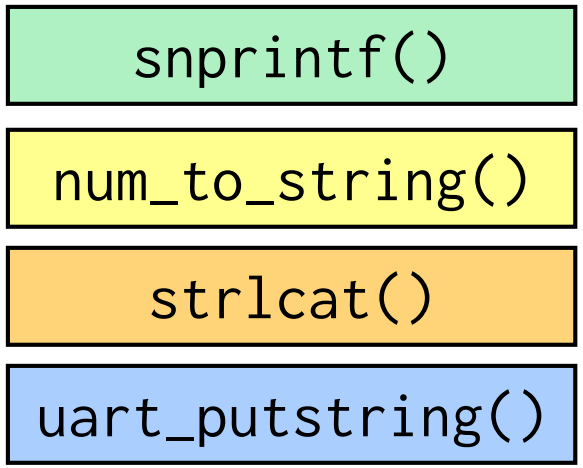
Module example: `printf`

Interface is single feature— make formatted string

```
int printf(const char* format, ...);
```

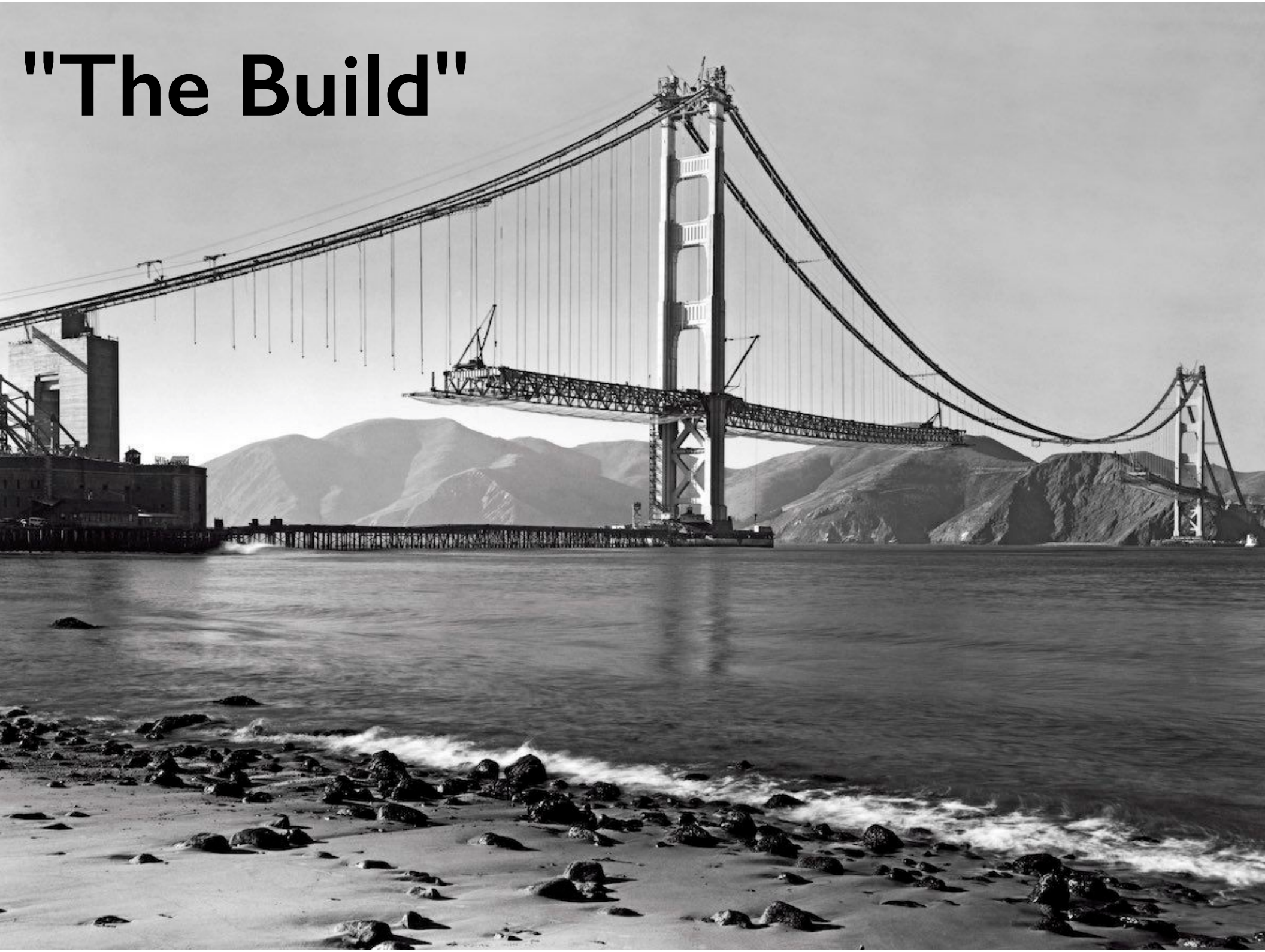
Implementation is complex, must:

- Parse format string
- Convert arguments into strings
- Concatenate conversions into output string
- Print output over uart



Decompose problem into smaller, simpler tasks, test independently, build up in layers

"The Build"



Lecture code examples

Link to code folder is on lecture schedule

code/clock

multi-module build (similar to assign2)

code/clock-libmango

clock program link with libmango

code/data

example of global data variables

Guide to binutils

as, ld, objcopy, ar

Tools for building program binaries

nm, size, strings, objdump

Tools for examining program binaries

NASA Command Center during SpaceX mission

CS107e Command Center: Makefile

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

ARCH    = -march=rv64im -mabi=lp64
ASFLAGS = $(ARCH)
CFLAGS  = $(ARCH) -Og -g -Wall -ffreestanding
LDFLAGS = -nostdlib -T memmap.ld
LDLIBS  =

all : $(APPLICATION).bin

%.bin: %.elf
   riscv64-unknown-elf-objcopy $< -O binary $@

%.elf: %.o $(MY_MODULES) cstart.o start.o          LINKER
   riscv64-unknown-elf-ld $(LDFLAGS) $^ $(LDLIBS) -o $@

%.o: %.c                                          COMPILER
   riscv64-unknown-elf-gcc $(CFLAGS) -c $< -o $@

%.o: %.s                                          ASSEMBLER
   riscv64-unknown-elf-as $(ASFLAGS) $< -o $@
```

Build process

PREPROCESSOR: cpp `program.c -> program.i`

Input: C source file

Output: replace **`#include`** and **`#define`** with text expansions

COMPILER: cc `program.i -> program.s`

Input: preprocessed C source file

Output: assembly file

ASSEMBLER: as `program.s -> program.o`

Input: assembly file

Output: machine code (binary-encoded)

LINKER: ld `program.o -> program.elf`

Input: multiple .o files, libraries

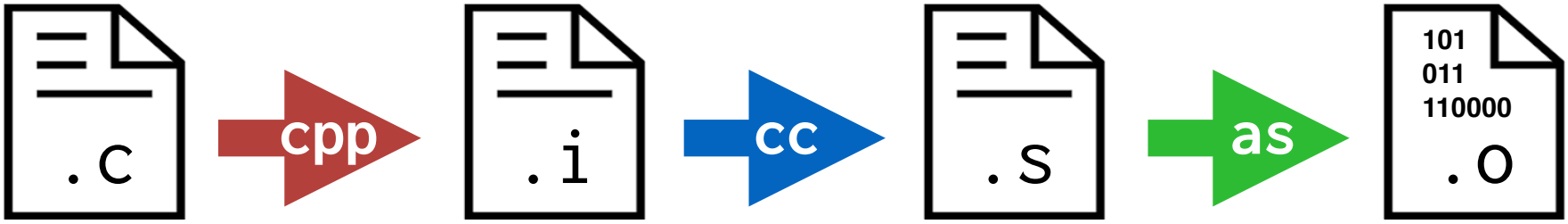
Output: executable in ELF

objcopy `program.elf -> program.bin`

Input: ELF executable

Output: raw binary

Compiling a single module



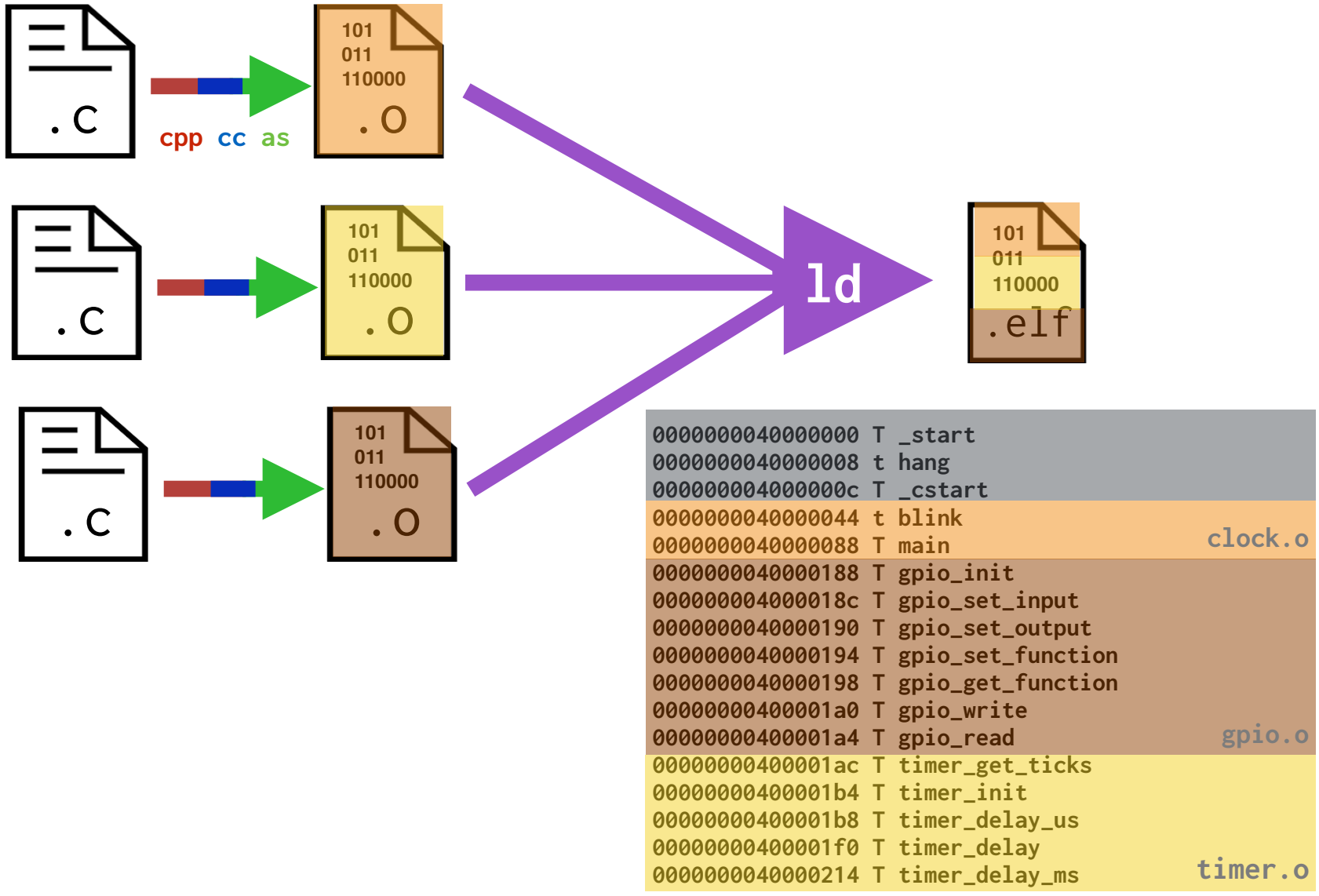
gcc -E gcc -S gcc -c

want to see intermediate files?

```
gcc --save-temps
```

Link modules into executable

Combines `clock.o gpio.o timer.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

C has single global namespace

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

Qualifier dictates symbol visibility and link behavior

- **`static`** private to module, not visible outside, no linking needed
- **`extern`** public, visible to other modules, cross-module references need link

Linker only concerned with **extern** symbols

Symbol resolution

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

Linker errors during symbol resolution:

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)

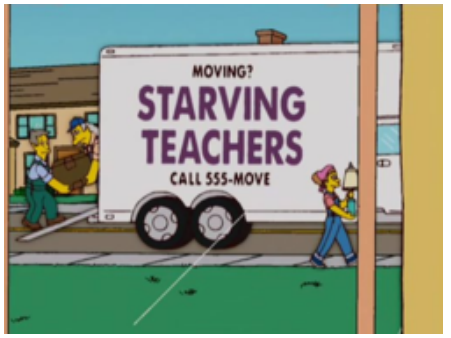
Linker processes .o files in order from left to right

Process each .o updates **Set U** and **Set D**

Continue until all .o processed

If end with **Set U** empty and **Set D** has no duplicates, link successful!

Symbol relocation



In .o file:

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

In executable file:

- modules consolidated in order, laid out end to end
- first module base address `0x40000000`
- base address of subsequent module adds size of all previous modules
- address of symbol = `module_base + symbol_offset_within_module`

Before link

start.s

```
_start:
    lui     sp,0x50000
    jal     _cstart
hang: j hang
```

```
$ riscv64-unknown-elf-objdump -d start.o
```

```
   0:  lui    sp,0x50000
   4:  jal    0
   8:  j      8 <hang>
```

Target address for `j hang` is 8, encoded as PC-relative offset (in this case, `pc+0`)

Target address for `jal _cstart` is 0 (placeholder!).
File `start.s` doesn't know where `_cstart` is! (Why not?)

Binutils: `objdump -d` shows disassembly

After link

```
$ riscv64-unknown-elf-objdump -d clock.elf
```

```
0000000040000000 <_start>:
    40000000:   lui sp,0x5000
    40000004:   jal 4000000c <_cstart>

0000000040000008 <hang>:
    40000008:   j  40000008 <hang>
```

All addresses have been relocated/finalized
Address of **_cstart** is now known — **4000000c**

The linker worked it out and replaced target placeholder with function's actual address

Handling of global data

Variable declared outside functions has **global** scope and **program** lifetime
(can be extern/static, read-only, initialized/not)

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

                              // initialized
int iNum = 0x1;
static int siNum = 0x22;

                              // const
const int cNum = 0x333;
static const int scNum = 0x4444;
```

Binutils: `size`, `nm`

```
$ riscv64-unknown-elf-size data.o
   text    data     bss     dec     hex filename
    377       8       8     393     189 data.o

$ riscv64-unknown-elf-nm -S data.o
0000000000000004 0000000000000004 R cNum
0000000000000004 0000000000000004 B gNum
0000000000000004 0000000000000004 D iNum
000000000000002c 00000000000000d4 T main
                 U printf
0000000000000000 0000000000000004 r scNum
0000000000000000 0000000000000004 b sgNum
0000000000000000 000000000000002c t show_var
0000000000000000 0000000000000004 d siNum
                 U uart_init
```

LEGEND

T/t = text (code)
D/d = read-write data
R/r = read-only data (**const**)
B/b = bss (*Block Started by Symbol*)
lowercase letter means **static**
uppercase **extern**

U = undefined

Libraries

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

Linker looks in library to find definitions for symbols in **Set U** (undefined). When symbol is found, entire module is linked in.

If linking this module results in new undefined symbols, they are added to **Set U**. Process repeats until no additional undefined symbols, move on to next library

Building with libmango

```
ARCH    = -march=rv64im -mabi=lp64
CFLAGS  = $(ARCH) -Og -I$$CS107E/include -Wall -ffreestanding
LDFLAGS = -nostdlib -L$$CS107E/lib -T memmap.ld
LDLIBS  = -lmango

%.bin: %.elf
    riscv64-unknown-elf-objcopy $< -O binary $@

%.elf: %.o
    riscv64-unknown-elf-ld $(LDFLAGS) $^ $(LDLIBS) -o $@

%.o: %.c
    riscv64-unknown-elf-gcc $(CFLAGS) -c $< -o $@
```

In Makefile for assigns 3 - 7,
`MY_MODULE_SOURCES` controls which of your modules to link,
otherwise will use from libmango

Diagnosing build errors

Which tool detects/reports… ?

mismatched type

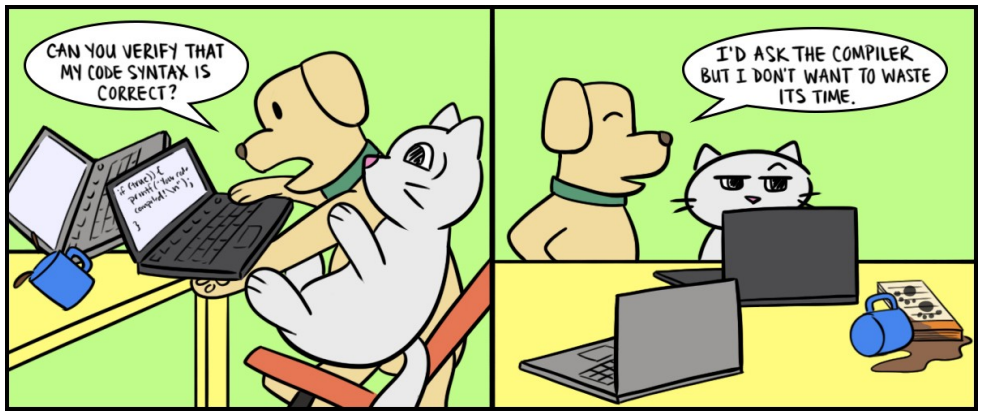
function call with wrong arguments

forgot #include

forgot to link with library

use variable uninitialized

two functions with same name



Modules and build process

Decompose into modules is skill to cultivate

- Separate system into coherent modules, minimize dependencies
- Clean abstractions, tidy interface
- Develop and test independently

Automate the build process

- Makefile is your command center
  - **Fast**: re-compile only what has changed
  - **Reliable**: track dependencies between modules
    - If file F changes, recompile everything that depends on F

Program start sequence

How is a program loaded into memory?

What must happens to start executing a program?

What is known about state of registers or contents of memory?

Does execution actually start at `main()`

👀 **Read our files!** 👀

`$CS107E/src/start.s`

`$CS107E/src/cstart.c`

`$CS107E/lib/memmap.ld`

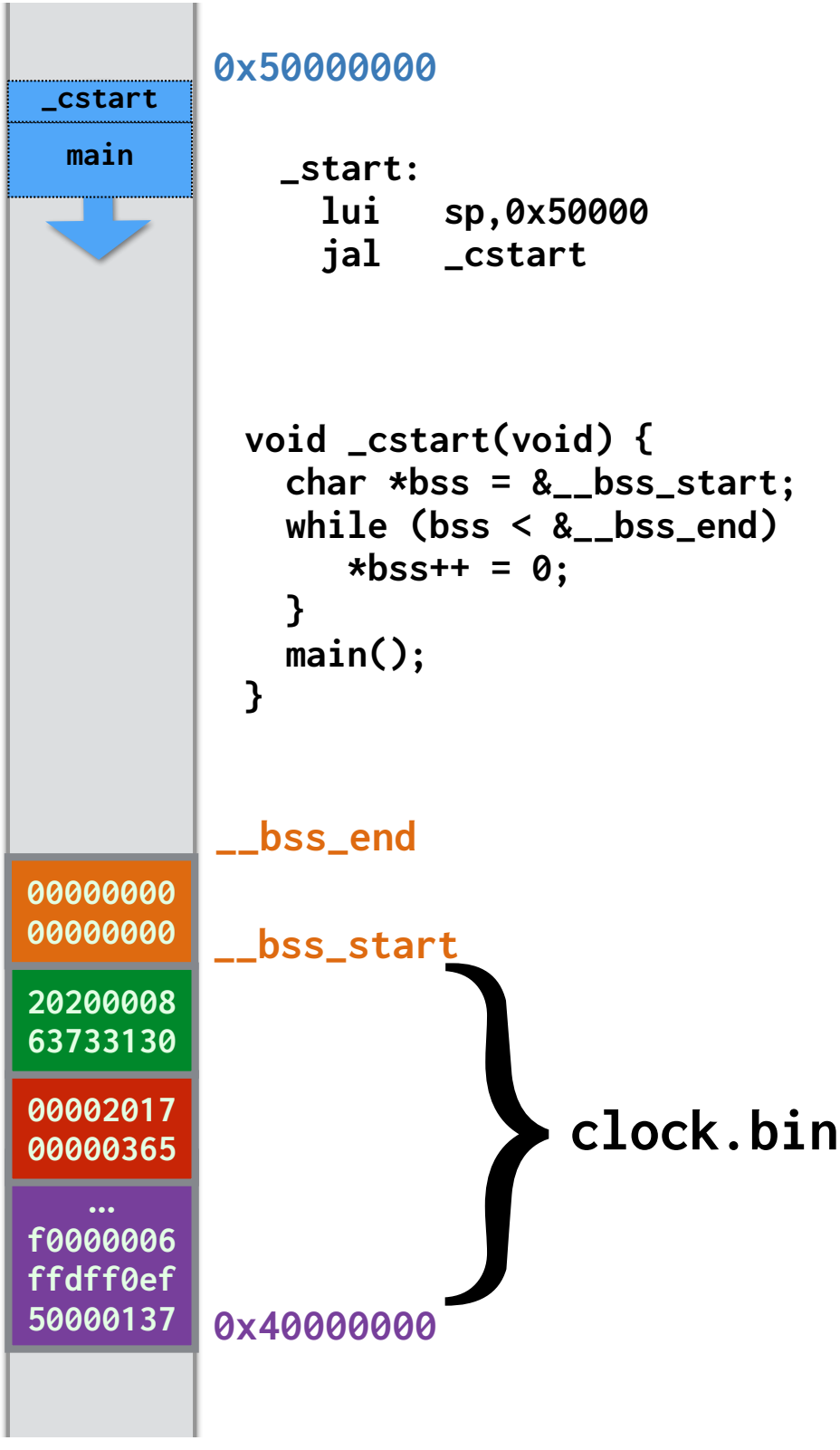
`memmap.ld` (linker script)

```
SECTIONS
{
    .text 0x40000000  :  { *(.text.start) *(.text*) }
    .rodata           :  { *(.rodata*) }
    .data             :  { *(.sdata*) }
    __bss_start       =  .;
    .bss              :  { *(.bss*) }
    __bss_end         =  .;
}
```

What is special about `.text.start` and why must it go first?

What is the significance of `0x40000000`?

```
SECTIONS
{
   .text 0x40000000 :{ *(.text.start)
                       *(.text*)}
   .rodata :          { *(.rodata*) }
   .data :            { *(.data*) }
   __bss_start        = .;
   .bss :             { *(.bss*) }
   __bss_end          = . ;
}
```



(zeroed data) .bss

(initialized data) .data

(read-only data) .rodata

.text

```
$ xfel write 0x40000000 clock.bin
$ xfel exec  0x40000000
```