# Admin

*No lecture Monday, MLK day*

*Assign 1 due Tuesday 5pm*
  Show off your bare-metal mettle!

*Pre-lab for lab2*
  Read gcc/make guides
  Read about 7-segment display
  Watch video of Ben Eater's mad breadboard skills


# Today: Hail the all-powerful C pointer

Addresses, pointers as abstractions for accessing memory
Memory layout for arrays and structs
Use of `volatile`

# From C to Assembly

C source describes computation at higher-level

- Portable abstractions (names, syntax, operators), consistent semantics
- Compiler emits asm for specific ISA/hardware
  - *major technical wizardry in back-end !*

Last lecture:

- C variable ⇒ registers
- C arithmetic/logical expression ⇒ ALU instructions
- C control flow ⇒ branch instructions

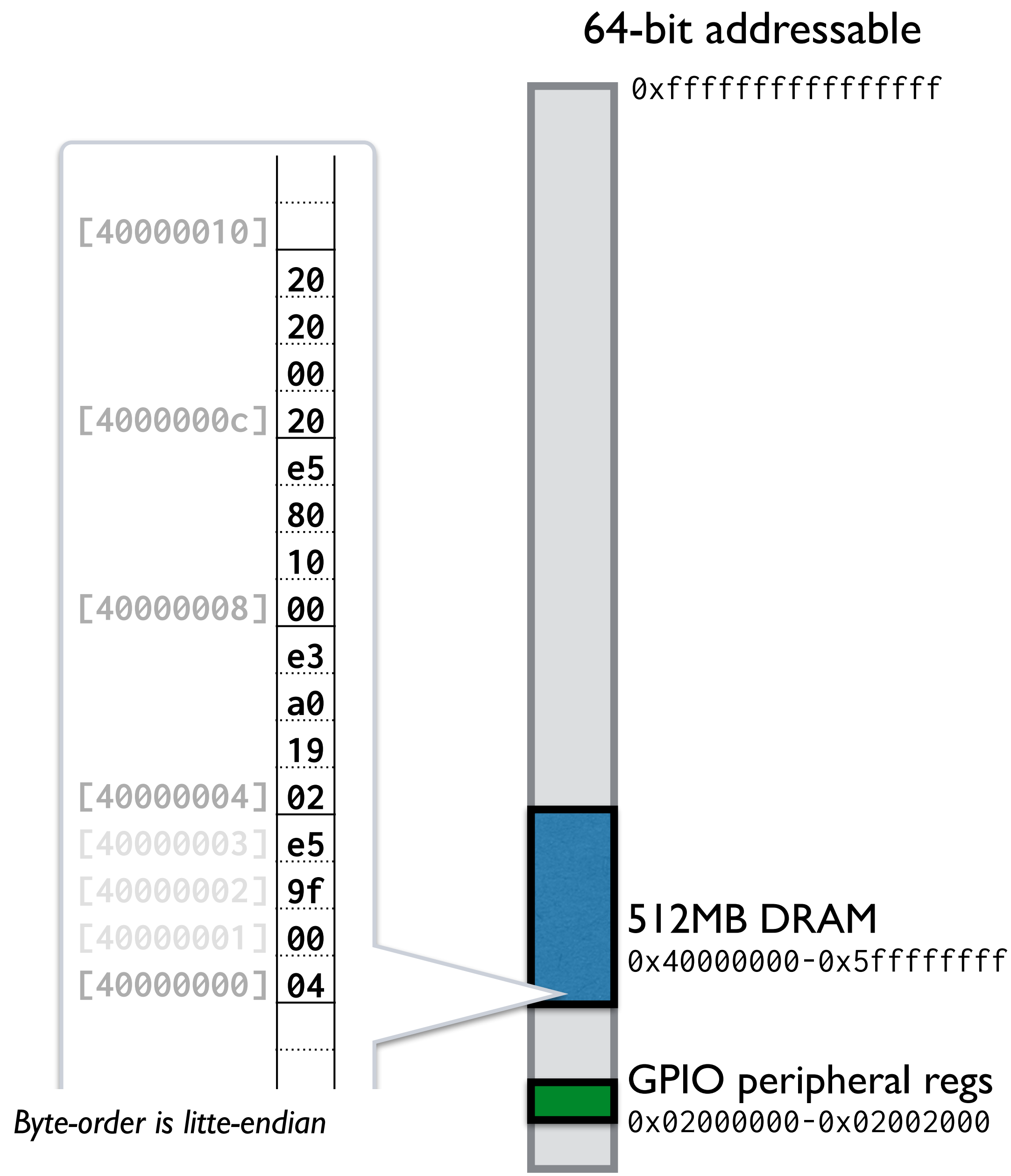
This lecture:

- C pointer ⇒ memory address
- Read/write memory ⇒ load/store instructions
- Array/struct data layout ⇒ address arithmetic

# Memory

Linear sequence of bytes, indexed by address

Instructions:
`lw` (load) from memory to register
`sw` (store) from register to memory



# Accessing memory in assembly

`lw` copies 4 bytes from memory address to register
`sw` copies 4 bytes from register to memory address

The memory address could be:

- location of a variable *or*
- location containing program instruction *or*
- memory-mapped peripheral *or* ...

The 4 bytes of data being copied could represent:

- a RISC-V instruction *or*
- an integer *or*
- 4 characters *or*
- bit pattern that controls peripheral *or* ...

```
lui     a0,0x2000
addi    a1,zero,1

sw      a1,0x30(a0)

sw      a1,0x40(a0)
```

And *assembly code doesn't care*

`lw` and `sw` simply access 4 bytes at memory address
No notion of "boundaries", agnostic to data type
Up to asm programmer to use correct address and respect type

C **pointers** (+ type system!) improved abstraction for accessing memory

# Pointer vocabulary

An *address* is a memory location. Address represented as `unsigned long (64-bit)`

A *pointer* is a variable that holds an address

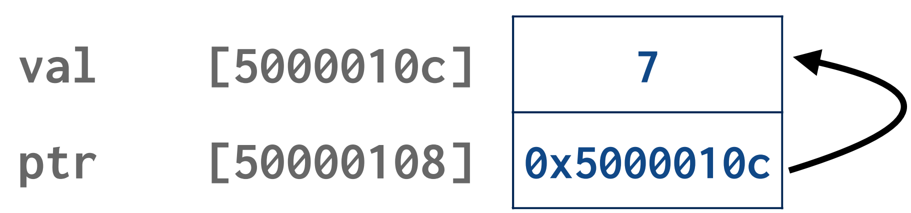
The "*pointee*" is the data stored at that address

`*` is the *dereference* operator, & is *address-of*

**C code**

```
int val = 5;
int *ptr = &val;
*ptr = 7;
```

**Memory**



# C pointer types

C enforces *type system:* every variable declares data type

- Reserve appropriate number of bytes
- Constrain operations to what is legal for type

Operations must respect data type

- Can't multiply two **`int*`** pointers, can't deference an **`int`**

C pointer variables distinguished by type of pointee

- Dereferencing an **`int*`** pointer accesses **`int`**
- Dereferencing a **`char*`** pointer accesses **`char`**
- Co-mingling pointers of different type disallowed

# What can C pointers buy us?

- Access data at specific address, e.g. `PB_CFG0 0x2000030`
- Access data by its offset relative to other nearby data (array elements, struct fields)
  - Related data grouped together, organizes memory
- Guide/constrain memory access to respect data type
  - (Better, but pointers still fundamentally unsafe...)
- Efficiently refer to shared data, avoid redundancy/duplication
- Build flexible, dynamic data structures at runtime

CULTURE FACT:

IN CODE, IT'S NOT CONSIDERED RUDE TO POINT.

```
    lui     a0,0x2000
    addi    a1,zero,1
    sw      a1,0x30(a0)

loop:
    xori    a1,a1,1
    sw      a1,0x40(a0)

    lui     a2,0x3f00
delay:
    addi    a2,a2,-1
    bne     a2,zero,delay

    j       loop
```

blink.s → c_blink.c

***let's do it!***

```
    lui     a0,0x2000
    addi    a1,zero,1
    sw      a1,0x30(a0)
loop:
    xori    a1,a1,1
    sw      a1,0x40(a0)

    lui     a2,0x3f00
delay:
    addi    a2,a2,-1
    bne     a2,zero,delay

    j       loop
```

## What all have we gained?

```
void main(void) {
   unsigned int *PB_CFG0 = 0x2000030;
   unsigned int *PB_DATA = 0x2000040;

   *PB_CFG0 = 1;

   int state = 1;
   while (1) {
      state = state ^ 1;
      *PB_DATA = state;
      int c = 0x3f00000;
      while (--c != 0) ;
   }
}
```

# Memory layout of C types

- Data for aggregate type (array, string, struct) is laid out in contiguous memory
- Base address (pointer) identifies start location
- Access to individual array element or struct field is by relative location, at offset from base
- In this way, single pointer (+ knowledge of layout) gives access to entire array/struct - **neat**!

| | |
|---|---|
| [5000010c] | [3] |
| [50000108] | [2] |
| [50000104] | [1] |
| [50000100] | array[0] |

| | | | | |
|---|---|---|---|---|
| [5000031c] | .f | | | |
| [50000318] | .b | .c | .d | .e |
| [50000314] | struct.a | | | |

| | | | | |
|---|---|---|---|---|
| [5000042c] | Y | ! | | |
| [50000428] | H | A | P | P |

# C arrays

Array is sequence of homogenous elements in contiguous memory
No sophisticated array "object", no track length, no bounds checking

Declare array by specifying element type and count of elements
Compiler reserves memory of correct size starting at base address
Access to elements by index calculates location as offset from base

```
int nums[5] = {4, 0, 0, -1, 7};

nums[2] = 189;
```

| | |
|---|---|
| [50000110] | 7 |
| [5000010c] | -1 |
| [50000108] | 189 |
| [50000104] | 0 |
| [50000100] | 4 |

# C structs

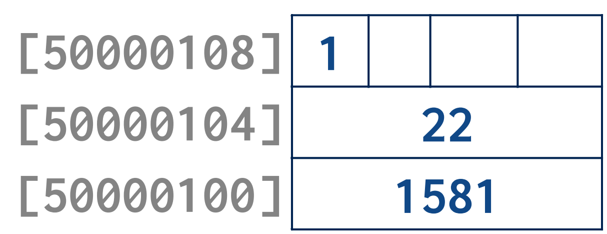
Struct is sequence of heterogenous fields in contiguous memory

`sizeof(struct) >= sum sizeof(fields)` (extra if padding)
Fields arranged in order of declaration
Access to field is offset from struct base

```
struct item {
   int sku;
   int price;
   bool in_stock;
};

struct item it;
it.sku = 1581;
it.in_stock = true;

struct item *ptr = &it;
ptr->price = 22; // (*it).price = 22;
```



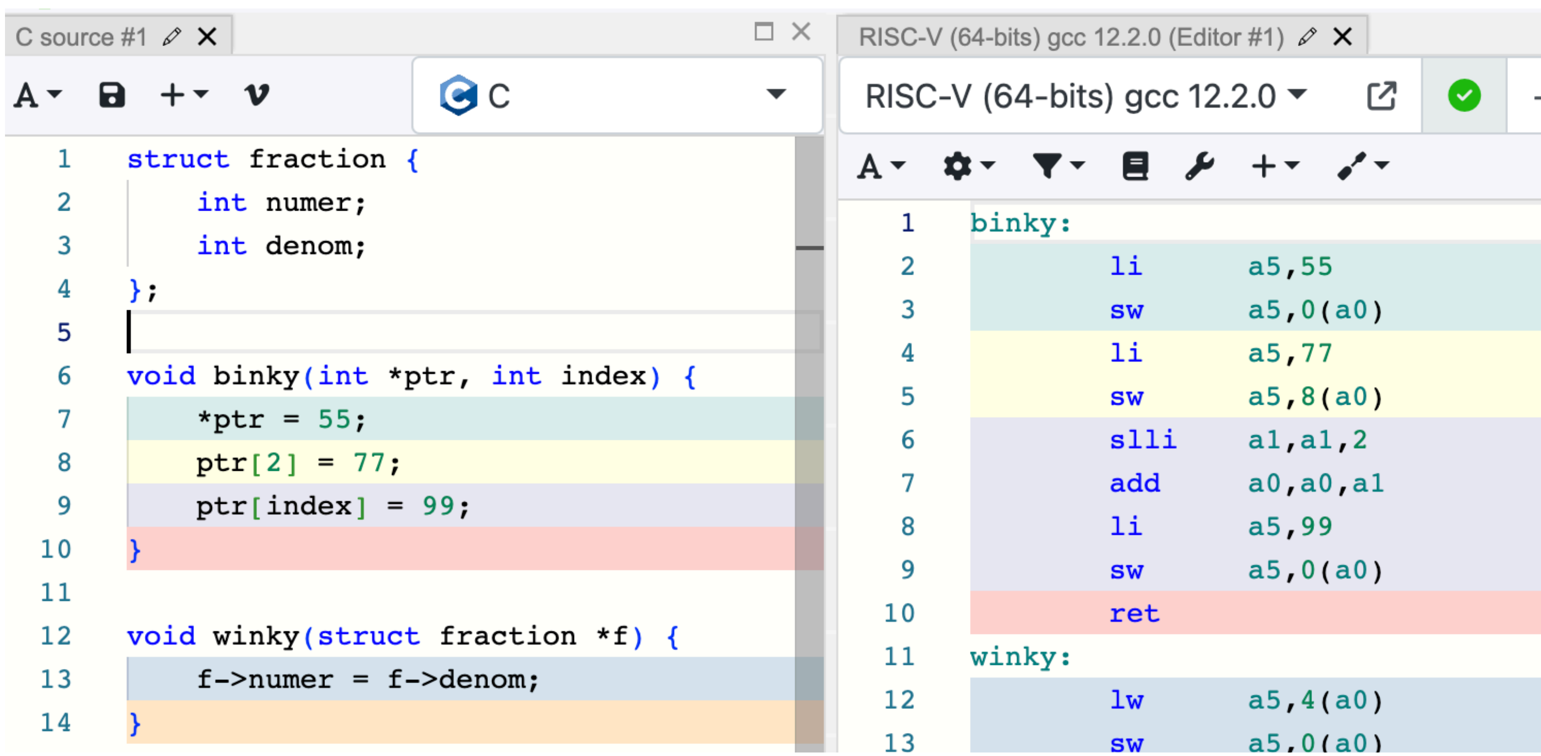
# Addresses in RISC-V

```
lw   a0, imm(a1)          // constant displacement
sw   a0, 0(a1)            // disp. can be zero
```

Load/store instructions have exactly one addressing mode: base address plus constant displacement

Any fancier address is built up via arithmetic ops

*Try CompilerExplorer to find out more!*



# Address arithmetic in C

Memory addresses can be manipulated arithmetically!

Address + offset to access data at neighboring location

```
unsigned int *base, *neighbor;

base = (unsigned int *)0x20000030;  // PB_CFG0
neighbor = base + 1;                // 0x20000034, PB_CFG1
```

*unsigned int is **4** bytes*

**IMPORTANT** ⚠️ ⚠️ ⚠️

C pointer add/subtract always **scaled** by `sizeof(pointee)`

e.g. operates in pointee-sized units

Array indexing is just pretty syntax for pointer arithmetic

```
array[index]   <=>   *(array + index)
```

# Pointers and structs

**9.7.4 Register List** *Ref: D1-H User Manual p.1093*

| Module Name | Base Address |
|---|---|
| GPIO | 0x02000000 |

| Register Name | Offset | Description |
|---|---|---|
| PB_CFG0 | 0x0030 | PB Configure Register 0 |
| PB_CFG1 | 0x0034 | PB Configure Register 1 |
| PB_DAT | 0x0040 | PB Data Register |
| PB_DRV0 | 0x0044 | PB Multi_Driving Register 0 |
| PB_DRV1 | 0x0048 | PB Multi_Driving Register 1 |
| PB_PULL0 | 0x0054 | PB Pull Register 0 |

```
struct gpio {
   unsigned int cfg[4];
   unsigned int data;
   unsigned int drv[4];
   unsigned int pull[2];
};
```

```
volatile struct gpio *pb = (struct gpio *)0x2000030;

pb->cfg[0] = ...
```

# The utility of pointers

## Accessing data by location is ubiquitous and powerful

### You learned in CS106B how pointers are useful

- Sharing data instead of redundancy/copying
- Construct linked structures (lists, trees, graphs)
- Dynamic/runtime allocation

### Now you see how it works under the hood

- Memory-mapped peripherals at fixed address
- Relative location to access struct fields and array elements

## What do we gain by using C pointers over raw `lw/sw`?

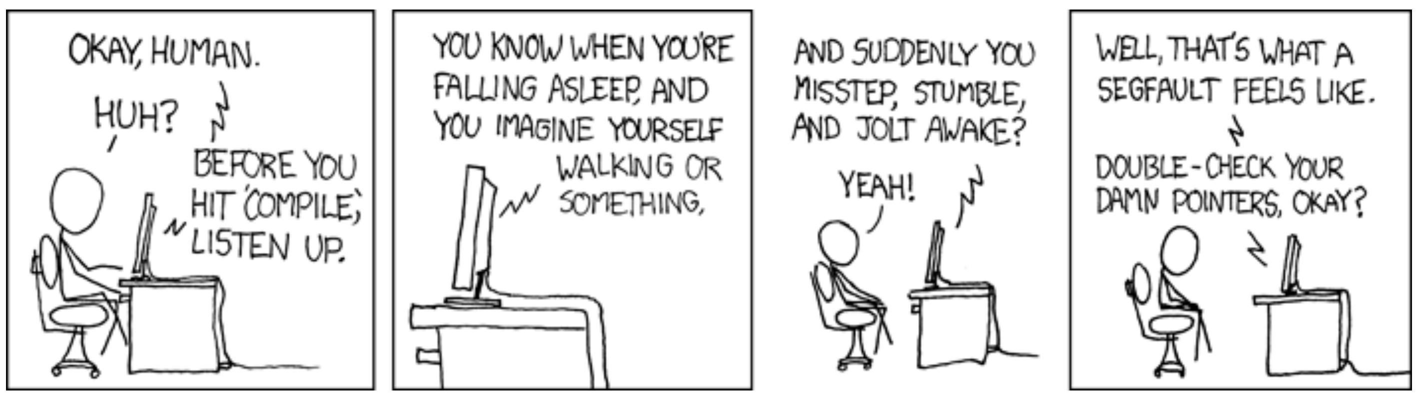
- Type system adds readability, some safety
- Pointee and level of indirection now explicit in the type
- Organize related data into contiguous locations, access using offset arithmetic

# Segmentation fault

**Pointers are ubiquitous in C, safety is low. Be vigilant!**

Q. For what reasons might a pointer be invalid?

Q. What is consequence of accessing invalid address
...in a hosted environment?
...in a bare-metal environment?

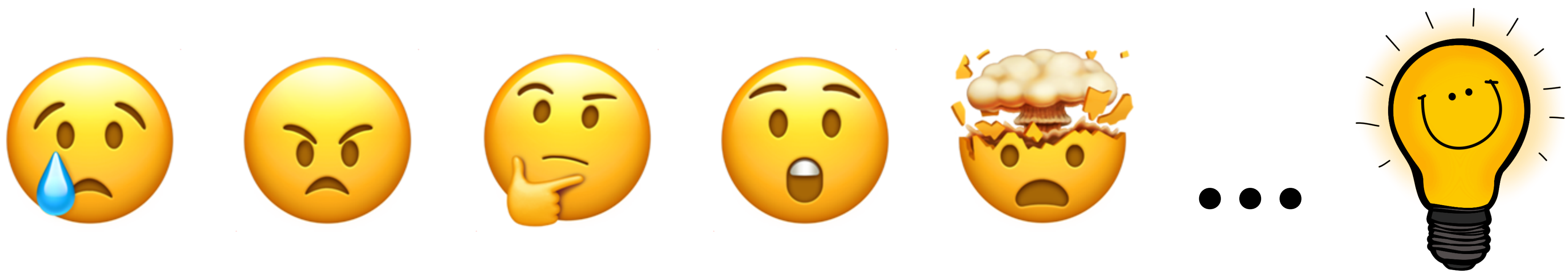



"The fault, dear Brutus, is not in our stars,
But in ourselves, that we are underlings."
[Julius Caesar (I, ii, 140-141)]

`c_button.c`

# The little button that wouldn’t

*A cautionary tale*



(or, why every systems programmer should be able to read assembly)

```
    lui     a0,0x2000
    addi    a1,zero,0x1
    sw      a1,0x30(a0)
    sw      zero,0x60(a0)

loop:
    lw      a2,0x70(a0)
    and     a2,a2,a1
    beq     a2,zero,off

on:
    sw      a1,0x40(a0)
    j       loop

off:
    sw      zero,0x40(a0)
    j       loop
```

```c
void main(void) {
  *PB_CFG0 = 1; // config PB0 output
  *PC_CFG0 = 0; // config PC0 input

   while (1) {
      if ((*PC_DATA & 1) != 0) {
         *PB_DATA = 1;
      } else {
         *PB_DATA = 0;
      }
   }
}
```

# Peripheral registers

These registers are mapped into address space of processor (memory-mapped IO).

These registers may behave **differently** than ordinary memory.

Peripheral registers access device state, and changing/reading that state may have complex effects beyond load/store of ordinary address.

*Q: What can happen when compiler makes assumptions reasonable for ordinary memory that* ***don't hold*** *for these oddball registers?*

# `volatile`

The compiler analyzes a code passage to determine where each variable is read/written. Generated assembly could be a literal translation of same steps or streamlined into equivalent sequence that has same effect. Neat!

But… if this memory location can be read/written externally (by another process, by peripheral), some optimizations may be invalid.

Qualifying a variable as **`volatile`** restricts compiler— it must not remove, coalesce, cache, or reorder accesses to this variable. The generated assembly must faithfully perform each access of the variable exactly as given in the C code.

*(If ever in doubt about what the compiler has done, use tools to review generated assembly and see for yourself...!)*