Admin

Assign 1 due Tuesday 5pm

Show off your bare-metal mettle!

Pre-lab for lab2

Read gcc/make guides

Read about 7-segment display

Today: Hail the all-powerful C pointer

Addresses, pointers as abstractions for accessing memory

Memory layout for arrays and structs

ARM addressing modes

Use of `volatile`

From C to Assembly

C language used to describe computation at high-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 ⇒ data processing instructions
- C control flow ⇒ condition codes, branch instructions, conditional execution

This lecture:

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

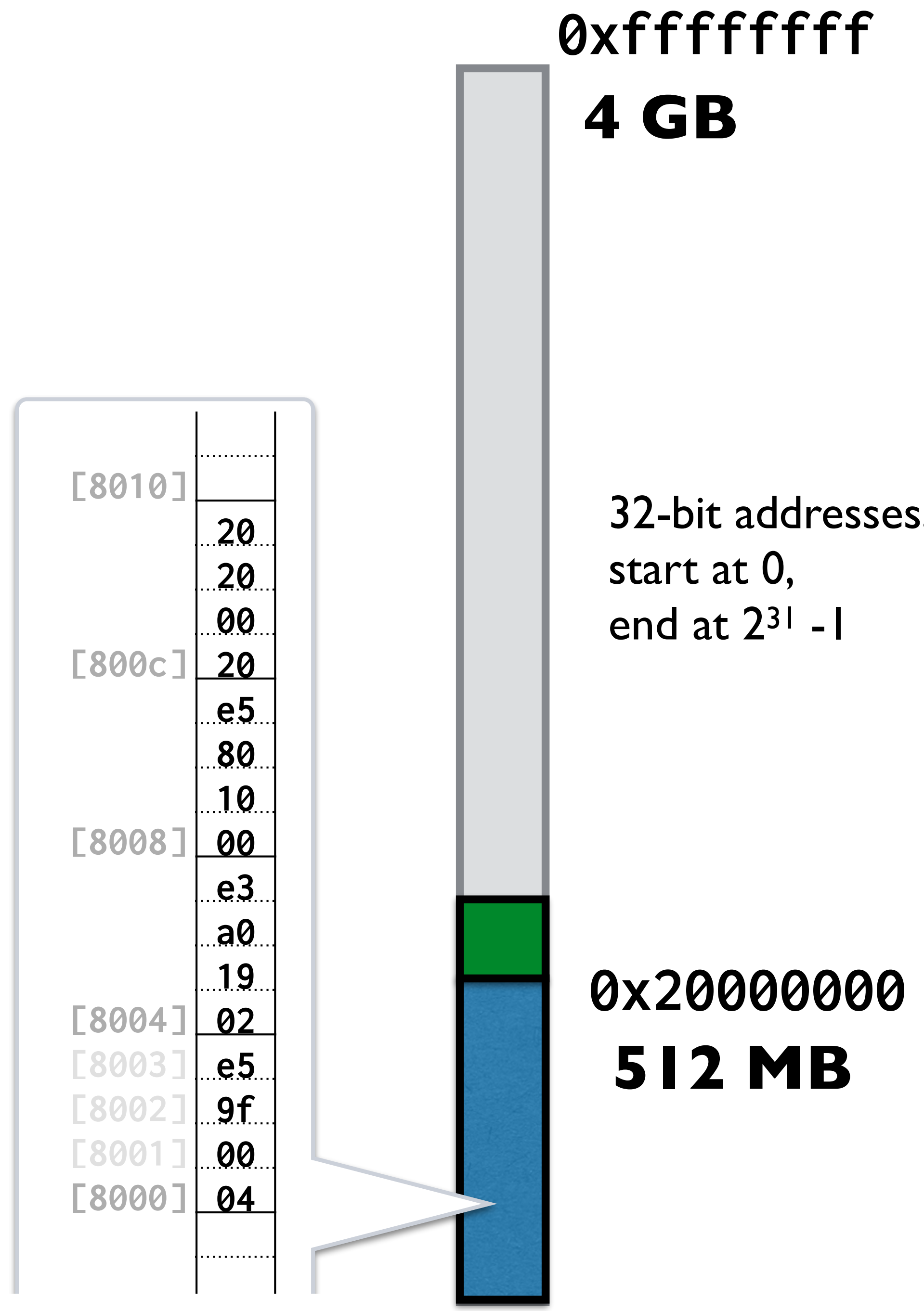
Memory

Linear sequence of bytes, indexed by address

Instructions:

`ldr` (load) from memory to register

`str` (store) from register to memory



Byte-order is litte-endian

Accessing memory in assembly

`ldr` and `str` copy 4 bytes from memory location to register (or vice versa)

The memory address could refer to:

- location reserved for a global or local variable *or*
- location containing program instruction *or*
- memory-mapped peripheral *or* ...

The 4 bytes of data being copied could represent:

- an address *or*
- an ARM instruction *or*
- an integer *or*
- 4 characters *or* ...

```
FSEL2: .word 0x20200008
SET0:  .word 0x2020001C

ldr r0, FSEL2
mov r1, #1
str r1, [r0]

ldr r0, SET0
mov r1, #(1<<20)
str r1, [r0]
```

`ldr` and `str` access memory location by address
No notion of "boundaries", agnostic to data type
Up to asm programmer to use correct address and respect type

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

Pointer vocabulary

An *address* is a memory location. Address represented as `unsigned int (32-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

val	[810c]	7
ptr	[8108]	0x0000810c

C pointer types

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

- Declaration used by compiler to reserve proper amount of space; determines what operations are legal for that data

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
- Generic **`void*`** pointer, raw address of indeterminate pointee type

```
ldr r0, FSEL2
mov r1, #1
str r1, [r0]

mov r1, #(1<<20)
ldr r0, SET0
str r1, [r0]

loop: b loop

FSEL2:  .word 0x20200008
SET0:   .word 0x2020001C
```

on.s ➜ c_on.c

let's do it!

What do C pointers buy us?

- Access data at specific address, e.g. FSEL2
- 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.

C arrays

Array is simply sequence of elements stored 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 is relative to base

```
char letters[4];
int nums[5];

letters[0] = 'a';
letters[3] = 'c';

nums[2] = 0x107e;
```

<table>
<tr><td>[8118]</td><td>61</td><td>?</td><td>?</td><td>63</td></tr>
<tr><td>[8114]</td><td colspan="4">?</td></tr>
<tr><td>[8110]</td><td colspan="4">?</td></tr>
<tr><td>[810c]</td><td colspan="4">0000107e</td></tr>
<tr><td>[8108]</td><td colspan="4">?</td></tr>
<tr><td>[8104]</td><td colspan="4">?</td></tr>
</table>

Address arithmetic

Memory addresses can be manipulated arithmetically!

Arithmetic used to access data at neighboring location

```
unsigned int *base, *neighbor;

base = (unsigned int *)0x20200000; // FSEL0
neighbor = base + 1;               // 0x20200004, FSEL1
```

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 arrays

```
int n, arr[4], *p;

p = arr;
p = &arr[0];      // same as prev line

arr = p;          // ILLEGAL, why?

*p = 3;
p[0] = 3;         // same as prev line

n = *(arr + 1);
n = arr[1];       // same as prev line
```

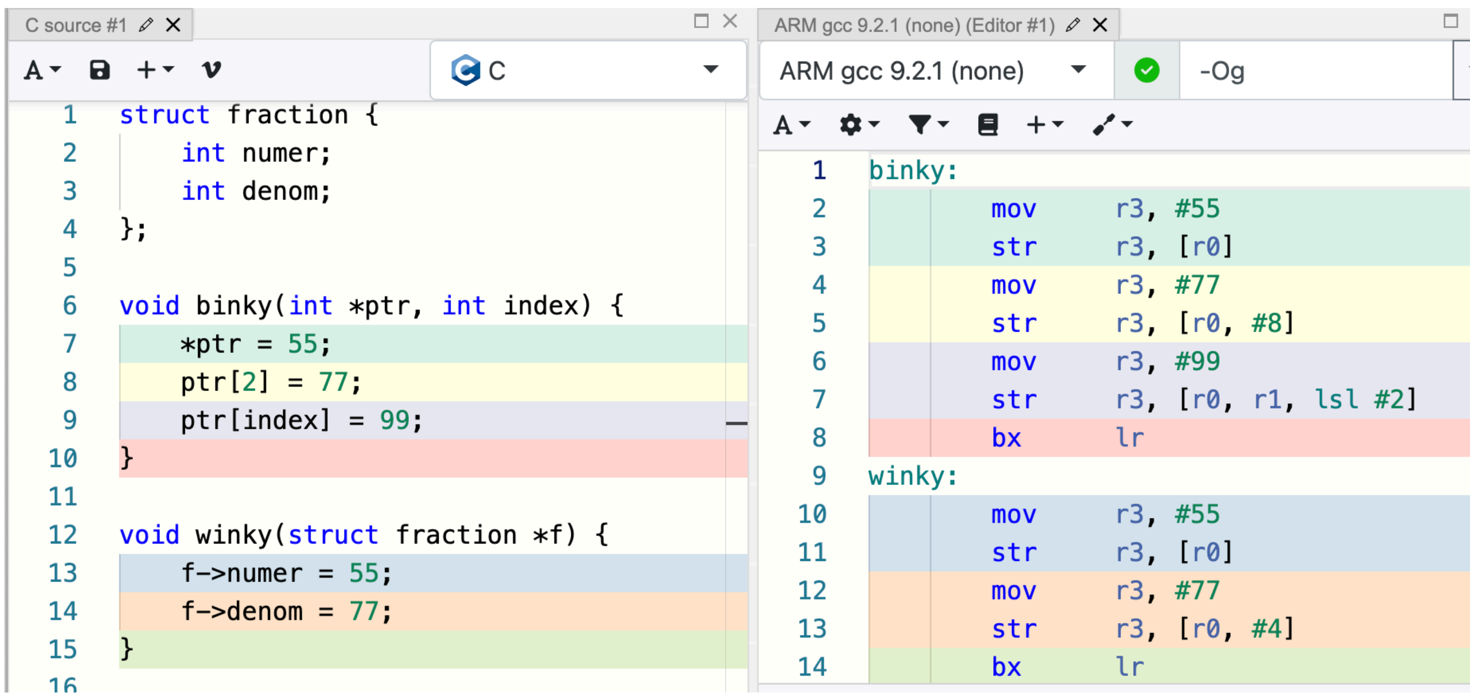
Fancy ARM addressing modes

```
ldr r0, [r1, #4]          // constant displacement
ldr r0, [r1, r2]          // variable displacement
ldr r0, [r1, r2, lsl #2]  // scaled index displacement
```

(Even fancier variants add pre/post update to move pointer along)

Consider how these relate to accessing C data types!

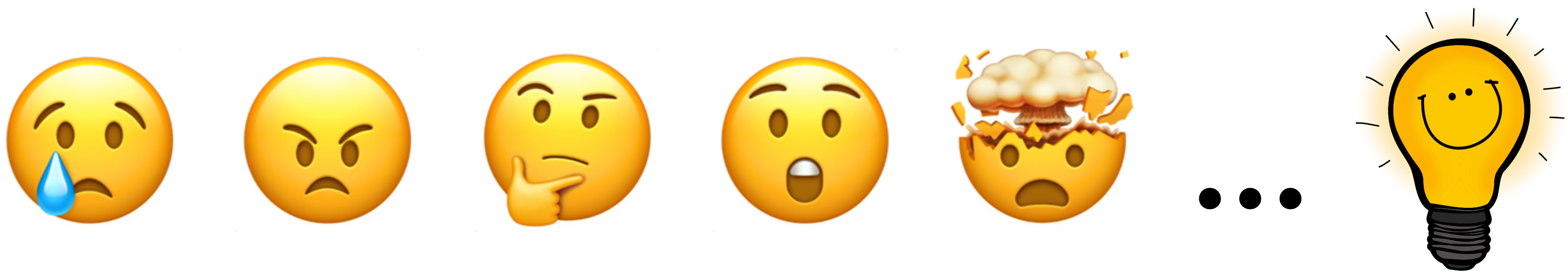
Use CompilerExplorer to find out more!



`c_button.c`

The little button that wouldn't

A cautionary tale



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

(Code available in courseware repo **`lectures/C_Pointers/code`***)*

```
    ldr r0, FSEL1   // config GPIO 10 as input
    mov r1, #0
    str r1, [r0]

    ldr r0, FSEL2   // config GPIO 20 as output
    mov r1, #1
    str r1, [r0]

    mov r2, #(1<<10)    // bit 10
    mov r3, #(1<<20)    // bit 20

    ldr r0, SET0
    str r3, [r0]   // set GPIO 20 (LED on)

wait:
    ldr r0, LEV0
    ldr r1, [r0]   // read LEV0
    tst r1, r2     // test bit 10
    bne wait       // if button not pressed, keep waiting

    ldr r0, CLR0
    str r3, [r0]   // clear GPIO 20 (LED off)


FSEL1: .word 0x20200004
FSEL2: .word 0x20200008
 SET0: .word 0x2020001C
 CLR0: .word 0x20200028
 LEV0: .word 0x20200034
```

```
void main(void) {

  *FSEL1 = 0; // config GPIO 10 as input (button)

  *FSEL2 = 1; // config GPIO 20 as output (LED)

  *SET0 = 1 << 20; // set GPIO 20 (LED on)

  while ((*LEV0 & (1 << 10)) != 0) // while not press
    ;                              // wait

  *CLR0 = 1 << 20; // clear GPIO 20 (LED off)
}
```

Peripheral registers

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

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

For example: Writing a 1 bit into SET register sets output to 1; writing a 0 bit into SET register has no effect. Writing a 1 bit into CLR sets the output to 0; writing a 0 bit into CLR has no effect. Neither SET or CLR can be read. To read the current value, access the LEV (level) register.

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

`volatile`

The compiler analyzes code to see where a variable is read/ written. Rather than execute each access literally, may streamline into an equivalent sequence that accomplishes same result. Neat!

If memory location can be read/written externally (by another process, by peripheral), these optimizations can be invalid!

Tagging a variable with **volatile** qualifier tells compiler that it cannot 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...!)

Pointers and structs

```
struct gpio {
  unsigned int fsel[6];
  unsigned int reservedA;
  unsigned int set[2];
  unsigned int reservedB;
  unsigned int clr[2];
  unsigned int reservedC;
  unsigned int lev[2];
};
```

Address	Field Name	Description	Size	Read/ Write
0x 7E20 0000	GPFSEL0	GPIO Function Select 0	32	R/W
0x 7E20 0000	GPFSEL0	GPIO Function Select 0	32	R/W
0x 7E20 0004	GPFSEL1	GPIO Function Select 1	32	R/W
0x 7E20 0008	GPFSEL2	GPIO Function Select 2	32	R/W
0x 7E20 000C	GPFSEL3	GPIO Function Select 3	32	R/W
0x 7E20 0010	GPFSEL4	GPIO Function Select 4	32	R/W
0x 7E20 0014	GPFSEL5	GPIO Function Select 5	32	R/W
0x 7E20 0018	-	Reserved	-	-
0x 7E20 001C	GPSET0	GPIO Pin Output Set 0	32	W
0x 7E20 0020	GPSET1	GPIO Pin Output Set 1	32	W
0x 7E20 0024	-	Reserved	-	-
0x 7E20 0028	GPCLR0	GPIO Pin Output Clear 0	32	W
0x 7E20 002C	GPCLR1	GPIO Pin Output Clear 1	32	W
0x 7E20 0030	-	Reserved	-	-
0x 7E20 0034	GPLEV0	GPIO Pin Level 0	32	R
0x 7E20 0038	GPLEV1	GPIO Pin Level 1	32	R

```
volatile struct gpio *gpio = (struct gpio *)0x20200000;

gpio->fsel[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 located at fixed address
- Access to struct fields and array elements using relative location

What do we gain by using C pointers over raw `ldr/str`?

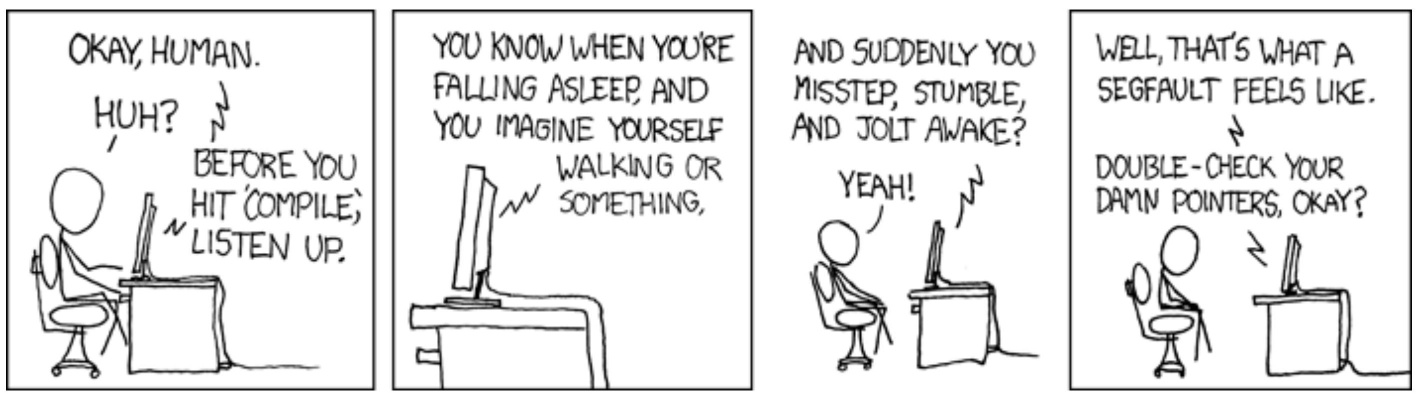
- Type system adds readability, some safety
- Pointee and level of indirection now explicit in the type
- Organize related data into contiguous locations, access using offset arithmetic

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)