

Admin

Assign 1 due Tuesday 5pm

Woo hoo, you did it!

Pre-lab for lab2

Read gcc/make guides

Preview 7-segment display in lab writeup

We're in person! Tue/Wed eve Gates B02



Today: Hail the all-powerful C pointer

Addresses, pointers as abstractions for accessing memory

Use of `volatile`

Implementation of arrays and structs

ARM addressing modes

Build Process

assembler, compiler, makefiles, bare metal

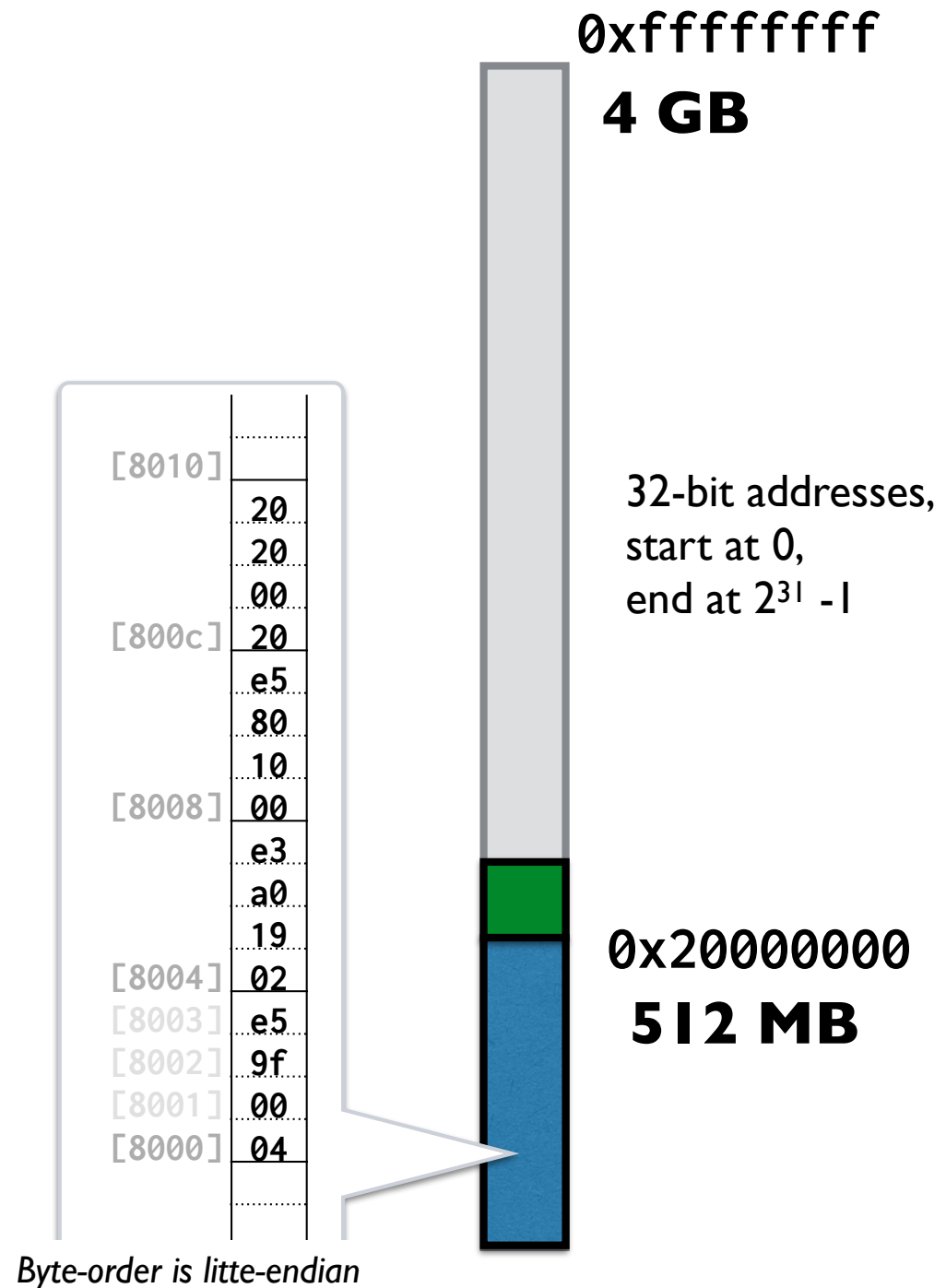
(See slides at end of previous lecture)

Memory

Linear sequence of bytes,
indexed by address

Program instructions and
data are stored in memory

Assembly instructions **ldr**
(load) from memory into
register to use, **str** (store)
updated value from
register back to memory



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* ...

Those 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", completely agnostic to data type

Ensure correct address/type is cognitive load for assembly programmer

C type system and pointers are improved abstraction for accessing memory

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)
 - Storing related data in related locations organizes use of 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.



Pointer vocabulary

An *address* is a memory location. Representation is unsigned 32-bit int.

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;
```

val [810c]

ptr [8108]

Memory



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

Pointer operations: & *

```
int m, n, *p, *q;
```

```
p = &n;
```

```
*p = n;           // same as prev line?
```

```
q = p;
```

```
*q = *p;          // same as prev line?
```

```
p = &m, q = &n;
```

```
*p = *q;
```

```
m = n;           // same as prev line?
```



```
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)
```

button.s



c_button.c

let's do it!

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
```

`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`)

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 sees in code where each variable is read/written. Rather than execute each access literally, may streamline into an equivalent sequence that accomplishes same result. Neat!

But, if variable may 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. 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...!)

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;
```

[8118]	61	?	?	63
[8114]	?			
[8110]	?			
[810c]	0000107e			
[8108]	?			
[8104]	?			

Address arithmetic

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
 $\text{array}[\text{index}] \quad \Leftrightarrow \quad *(\text{array} + \text{index})$

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
```

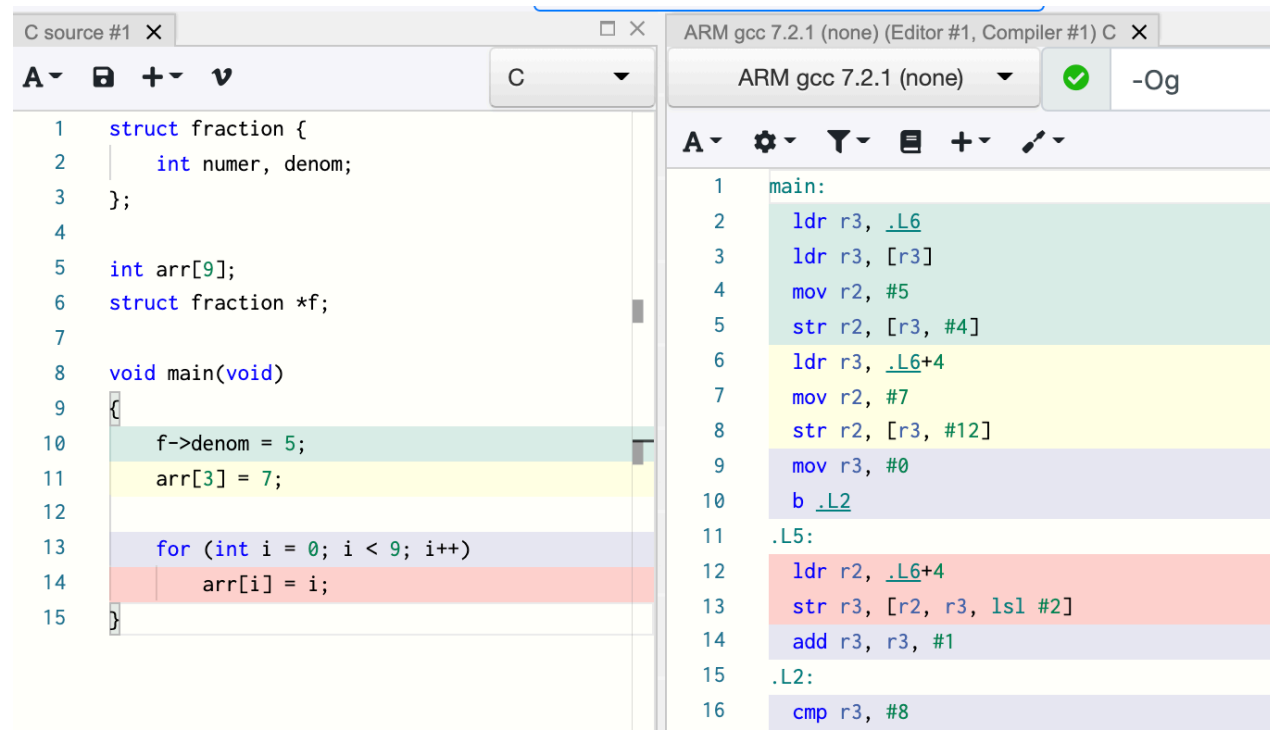
Fancy ARM addressing modes

```
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)

Q: How do these relate to accessing data structures in C?

Try CompilerExplorer to find out!



The screenshot shows the Compiler Explorer interface with two panes. The left pane displays the C source code for a program that defines a `fraction` struct, an array `arr`, and a `main` function that initializes `f->denom`, `arr[3]`, and then iterates over `arr` from index 0 to 8. The right pane shows the corresponding ARM assembly code generated by GCC 7.2.1. The assembly includes instructions for loading and storing values, moving registers, and a loop that uses the scaled index addressing mode `ldr r2, [r2, r3, lsl #2]` to access elements of the `arr` array.

```
C source #1 X
1 struct fraction {
2     int numer, denom;
3 };
4
5 int arr[9];
6 struct fraction *f;
7
8 void main(void)
9 {
10     f->denom = 5;
11     arr[3] = 7;
12
13     for (int i = 0; i < 9; i++)
14         arr[i] = i;
15 }

ARM gcc 7.2.1 (none) (Editor #1, Compiler #1) C X
ARM gcc 7.2.1 (none) -Og
1 main:
2     ldr r3, .L6
3     ldr r3, [r3]
4     mov r2, #5
5     str r2, [r3, #4]
6     ldr r3, .L6+4
7     mov r2, #7
8     str r2, [r3, #12]
9     mov r3, #0
10    b .L2
11 .L5:
12    ldr r2, .L6+4
13    str r3, [r2, r3, lsl #2]
14    add r3, r3, #1
15 .L2:
16    cmp r3, #8
```


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 by 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 \(I, ii, 140-141\)](#)