

Admin

Assign 1 due tomorrow

Congrats on proving
your bare-metal mettle!

Pre-lab for lab2

Read about make, 7-segment display
Bring your tools if you have them!



Today: Hail the all-powerful C pointer

Makefiles, bare metal build process

Addresses, pointers as abstractions for accessing memory

Use of `volatile`

Implementation of arrays and structs

ARM addressing modes

Build process

(See slides at end of previous lecture)

Compile-time vs. runtime

Compile-time: compiler is running on your laptop

- reads your C code, parse/check semantically valid
- analyzes code to understand structure/intent
- generates assembly instructions, creates program binary

Runtime: program binary is running on Pi

- all that remains is generated assembly instructions
- fetch/decode/execute cycle

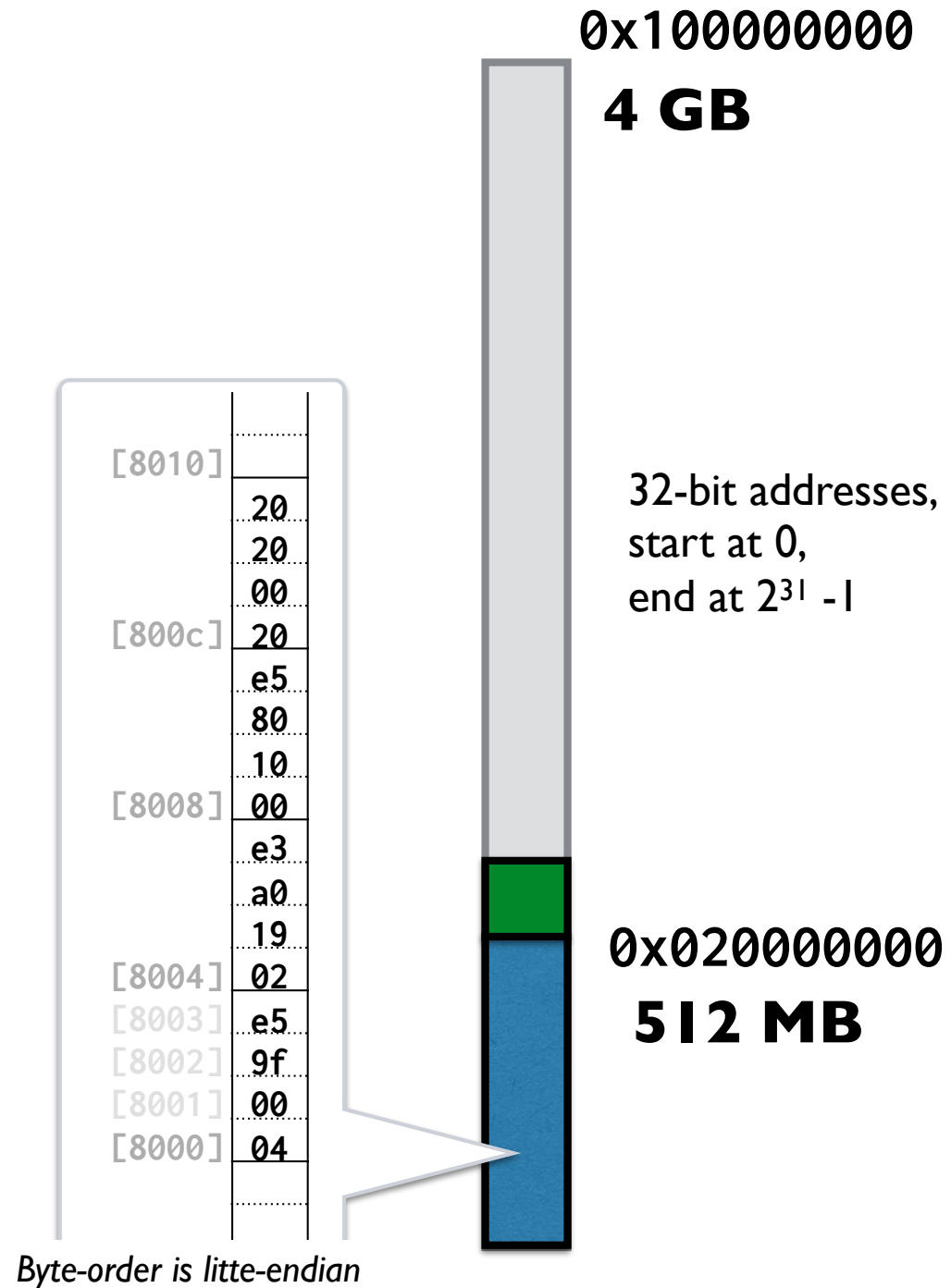
The work optimizer does at CT is intended to streamline number of instructions to be executed at RT

Memory

Linear sequence of bytes,
indexed by address

Program instructions and
data are stored in memory

Assembly instructions **load**
from memory into register
to use, **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:

- a location reserved for a global or local variable or
- a location containing program instructions or
- a memory-mapped peripheral or
- an unused/invalid location or ...

Those 4 bytes of data being copied could represent:

- an address or
- an integer or
- 4 characters or
- an ARM instruction 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]
```

Assembly instructions allow access to any memory location by address

No notion of "boundaries", completely agnostic to data type

Correctly accessing memory is cognitive load for programmer

C type system and pointers are improved abstraction for accessing memory

What do C pointers buy us?

- Access specific memory by 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 // configure GPIO 10 for input
mov r1, #0
str r1, [r0]
```

```
ldr r0, FSEL2 // configure GPIO 20 for output
mov r1, #1
str r1, [r0]
```


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

```
loop:
    ldr r0, LEV0
    ldr r1, [r0] // read GPIO 10
    tst r1, r2
    beq on      // if button down, turn on LED, else turn off

    off:        // set GPIO 20 low
        ldr r0, CLR0
        str r3, [r0]
        b loop

    on:         // set GPIO 20 high
        ldr r0, SET0
        str r3, [r0]
        b loop
```

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

button.s  c_button.c

let's do it!

`c_button.c`

The little button that wouldn't

A cautionary tale



...



(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]	00000107e			
[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
```

NOTE: C pointer add/subtract is scaled by `sizeof(pointee)`
e.g. operates in pointee-sized units

Array indexing is just pretty syntax for pointer arithmetic

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
```


C-strings

No string "abstraction", just sequence of chars in memory, e.g. char array,
Must be terminated by null char (zero byte)
char* points to first character

Trace the following code and draw a memory diagram

```
char *s = "Leland";  
char *t;  
char buf[9];
```

```
t = s;  
s[0] = 'R';  
*t = 'Z';  
s = buf + 4;    // where does s point?  
s[1] = t[3];    // what value changes?
```

Address arithmetic

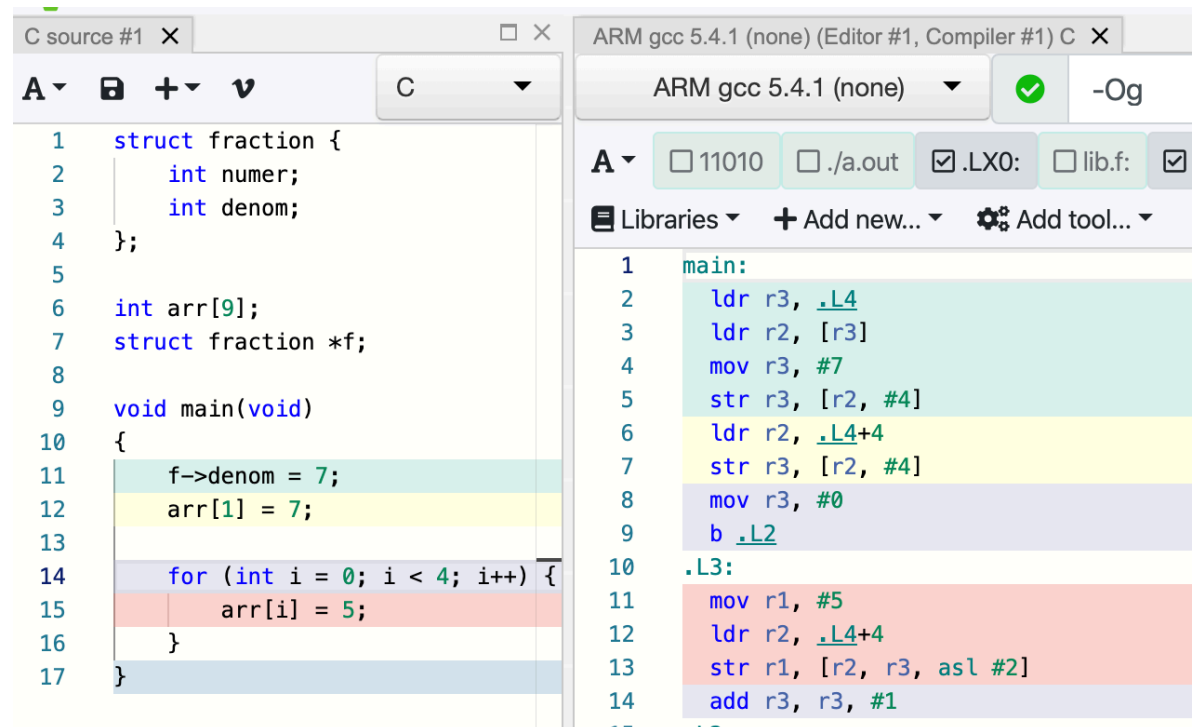
Fancy ARM addressing modes

```
ldr r0, [r1, #4]           // constant displacement
ldr r0, [r1, r2]           // variable displacement
ldr r0, [r1, r2, asl #3]   // 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. On the left, the C source code is displayed in a file named 'C source #1'. It defines a 'fraction' struct with 'numer' and 'denom' fields, an array 'arr' of size 9, and a 'main' function that initializes 'denom' to 7, sets 'arr[1]' to 7, and then iterates over 'arr' from index 0 to 3, setting each element to 5. On the right, the ARM assembly output is shown for 'ARM gcc 5.4.1 (none)'. The assembly includes instructions for loading and storing values, moving registers, and branching, corresponding to the C code's logic. The assembly is color-coded to match the C code's structure.

```
C source #1 X
1 struct fraction {
2     int numer;
3     int denom;
4 };
5
6 int arr[9];
7 struct fraction *f;
8
9 void main(void)
10 {
11     f->denom = 7;
12     arr[1] = 7;
13
14     for (int i = 0; i < 4; i++) {
15         arr[i] = 5;
16     }
17 }
```

```
ARM gcc 5.4.1 (none) (Editor #1, Compiler #1) C X
ARM gcc 5.4.1 (none) -Og
11010 .a.out .LX0: lib.f:
Libraries + Add new... Add tool...
1 main:
2     ldr r3, .L4
3     ldr r2, [r3]
4     mov r3, #7
5     str r3, [r2, #4]
6     ldr r2, .L4+4
7     str r3, [r2, #4]
8     mov r3, #0
9     b .L2
10 .L3:
11     mov r1, #5
12     ldr r2, .L4+4
13     str r1, [r2, r3, asl #2]
14     add r3, r3, #1
```

The utility of pointers

Accessing data by location is ubiquitous and powerful

You learned in previous course 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

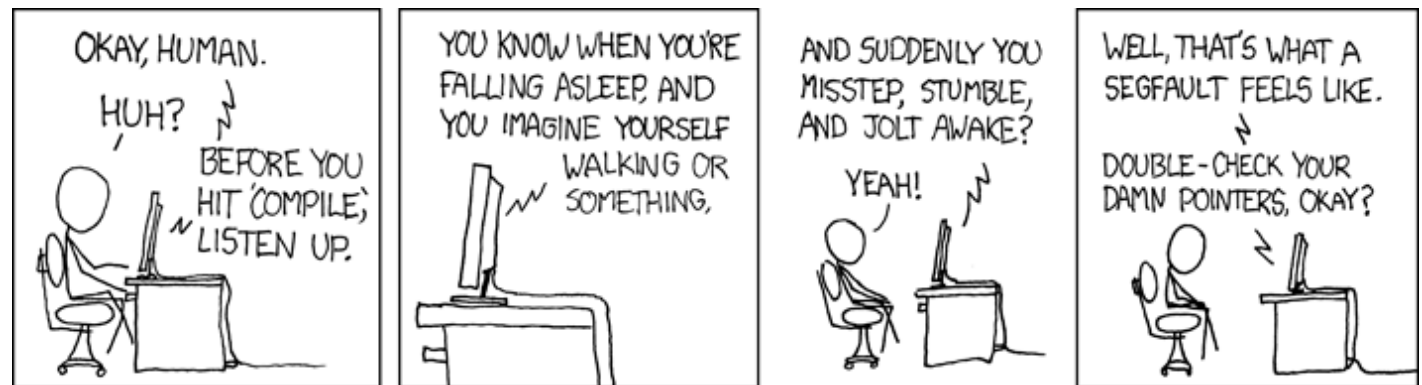
- Organize related data into contiguous locations, access using offset arithmetic

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