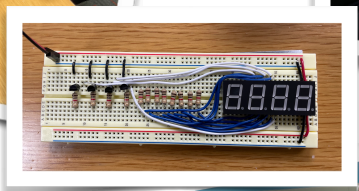
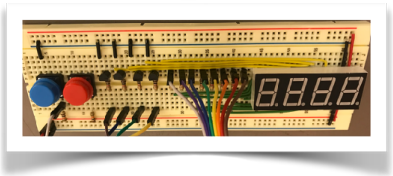
Admin

⭐ *Lab 2 triumph* 🏆

Assign 2, yay C 💪





Today: C functions

Implementation of C function calls

Management of runtime stack, register use

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 allocation, flexible, configurable at runtime

Now you see how it works under the hood

- Constant address access at fixed location: memory-mapped peripherals
- Pointer arithmetic to access at relative location: struct fields, array elements
  - (Also pc-relative access to branch target, data words)

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

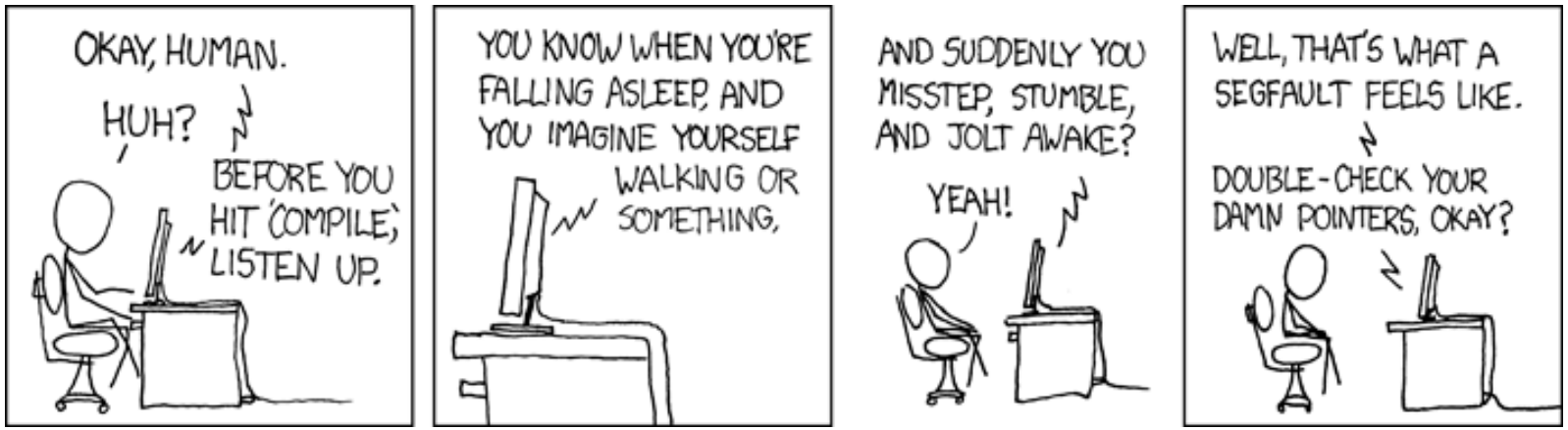
- Type system adds readability, some safety
- Pointee and level of indirection explicit in the type
- Organize related data into contiguous locations, access using offset arithmetic

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)

Excerpted from blink.s

```
loop:
    ldr r0, SET0
    str r1, [r0]

    mov r2, #DELAY
    wait1:
        subs r2, #1
        bne wait1

    ldr r0, CLR0
    str r1, [r0]

    mov r2, #DELAY
    wait2:
        subs r2, #1
        bne wait2

b loop
```

Sure seems same code, would be nice to unify…

```
loop:
  ldr r0, SET0
  str r1, [r0]
  b delay
  ldr r0, CLR0
  str r1, [r0]
  b delay
  b loop
```

```
delay:
  mov r2, #DELAY
  wait:
    subs r2, #1
    bne wait

// but... where to go now?
```

```
loop:
    ldr r0, SET0
    str r1, [r0]

    mov r14, pc
    b delay

    ldr r0, CLR0
    str r1, [r0]

    mov r14, pc
    b delay

    b loop
```

Recall this quirk of ARM:
when executing instruction at address N, pc is tracking N+8 due to pipelining fetch-decode-execute

```
delay:
    mov r2, #DELAY
    wait:
        subs r2, #1
        bne wait
    mov pc, r14
```

We've just invented our own link register!

```
loop:
    ldr r0, SET0
    str r1, [r0]

    mov r0, #DELAY
    mov r14, pc
    b delay

    ldr r0, CLR0
    str r1, [r0]

    mov r0, #DELAY >> 2
    mov r14, pc
    b delay

    b loop
```

```
delay:
wait:
    subs r0, #1
    bne wait
    mov pc, r14
```

We've just invented our own parameter passing!

Anatomy of C function call

```
int factorial(int n)
{
   int result = 1;
   for (int i = n; i > 1; i--)
      result *= i;
   return result;
}
```

Call and return

Pass arguments

Local variables

Return value

Scratch/work space

Complication: nested function calls, recursion

Application binary interface

ABI specifies how code interoperates:

- Mechanism for call/return
- How parameters passed
- How return value communicated
- Use of registers (ownership/preservation)
- Stack management (up/down, alignment)

`arm-none-eabi`

ARM architecture

no hosting OS

embedded ABI

Mechanics of call/return

Caller puts up to 4 arguments in `r0,r1,r2,r3`
Call instruction is `bl` (branch and link)

```
mov r0, #100
mov r1, #7
bl sum          // will set lr = pc-4
```

Callee puts return value in `r0`
Return instruction is `bx` (branch exchange)

```
add r0, r0, r1
bx lr           // pc = lr
```

btw: `lr` *is alias for* `r14`, `pc` *is alias for* `r15`

Caller and Callee

caller: function doing the calling

callee: function being called

`main` is caller of `range`

`range` is callee of `main`

`range` is caller of `abs`

```
void main(void) {
    range(13, 99);
}

int range(int a, int b) {
    return abs(a-b);
}

int abs(int v) {
    return v < 0 ? -v : v;
}
```

Register Ownership

`r0-r3` are **callee-owned** registers

- **Callee** can freely use/modify these registers
- **Caller** cedes to callee, has no expectation of register contents after call

`r4-r13` are **caller-owned** registers

- **Caller** retains ownership, expects register contents to be same after call as it was before call
- **Callee** cannot use/modify these registers unless takes steps to preserve/restore values

Discuss...

1. If callee needs scratch space for an intermediate result, which type of register should it choose?

2. Why might a callee need to use a caller-owned register? What does callee have to do if using one?

3. What is the advantage in having some registers callee-owned and others caller-owned? Wouldn't it be simpler if all treated the same?

The stack to the rescue!

Reserve section of memory to store data for executing function

Stack frame allocated per function invocation
Can store local variables, scratch values, saved registers

- LIFO: **`push`** adds value, **`pop`** removes lastmost value
- **`r13`** (alias **`sp`**) points to lastmost value pushed
- stack grows down
  - newer values at lower addresses
  - push subtracts from **`sp`**
  - pop adds to **`sp`**
- **`push`**/**`pop`** is nickname for "store/load (multiple) **`sp`**-relative with writeback" (**`stm`**/**`ldm`**)

```
// start.s
mov sp, #0x8000000
bl main
```

```
void main(void)
{
    delta(3, 7);
}

int delta(int a, int b)
{
  int diff = sqr(a) - sqr(b);
  return diff;
}

int sqr(int v)
{
  return v * v;
}
```

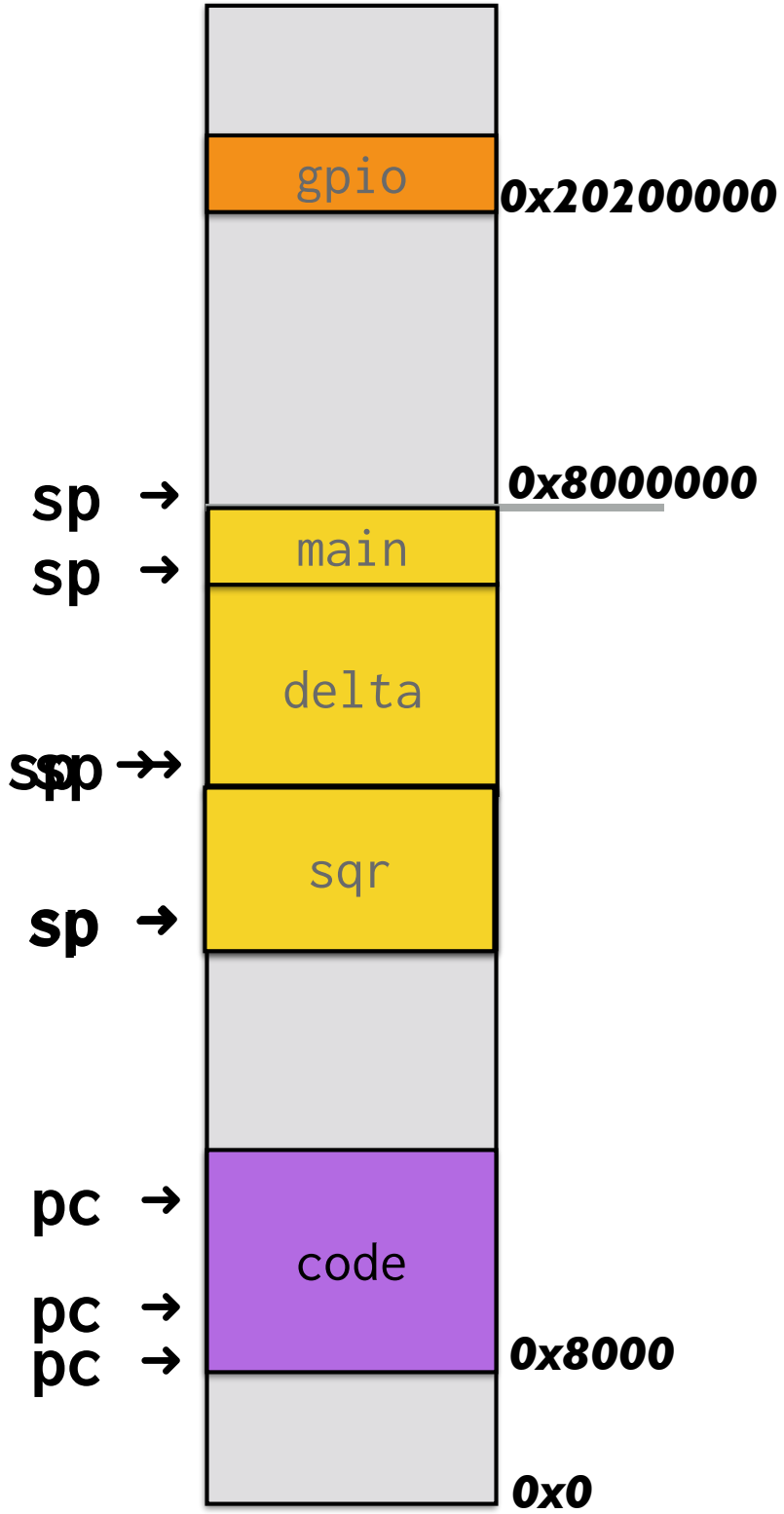


Diagram not to scale

Stack operations

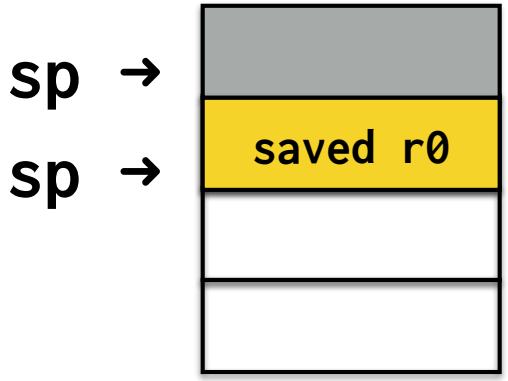
```
// push to saved reg val on stack
// *--sp = r0
// decrement sp before store
// equivalent:  str r0, [sp, #-4]!

push {r0}


// pop to restore reg val from stack
// r0 = *sp++
// increment sp after load
// equivalent: ldr r0, [sp], #4

pop {r0}
```

"Full Descending" stack



ARM ABI requires sp to be 8-byte aligned
so always push/pop even number of registers (2, 4, 6, ...)

Gdb debugger

Debugger is incredibly useful

Allows you to run your program in a monitored context

Can set breakpoints, examine state, change values, reroute control, and more

Running bare metal, we have no on-Pi debugger 😢

But, **gdb** has simulation mode where it pretends to be an ARM processor, running on your laptop 🙌

Pretty good approximation (not perfect, e.g. no peripherals)

Let's try it now!

Run under debugger and observe stack in action

```
$ arm-none-eabi-gdb program.elf
(gdb) target sim
(gdb) load
```

👀 **Read our course guide on gdb!** 👀

http://cs107e.github.io/guides/gdb/

C vs Assembly Smackdown

Why C?

Variable names, type system

Function decomposition, control flow

Portable abstractions

Consistent semantics

Compiler back-end doing heavy lifting - yay!

Why assembly?

Execution is always in asm, this is the real deal -- WYSIWYG

Ability to drop down and review/debug asm is key

Certain hardware features only accessible via asm

Hand-code in asm for optimization or obtain precise timing