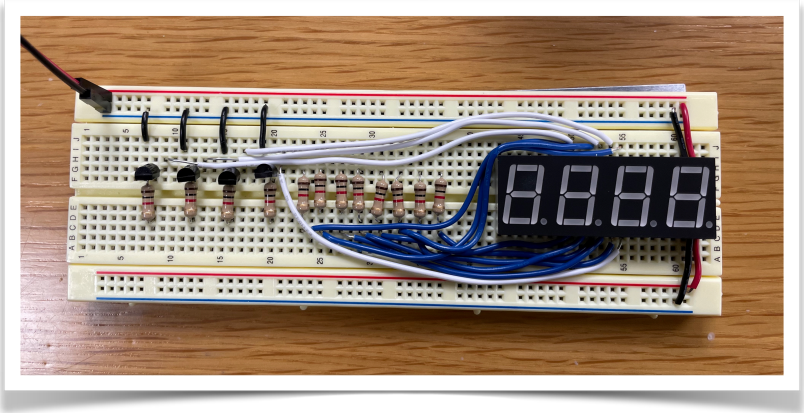
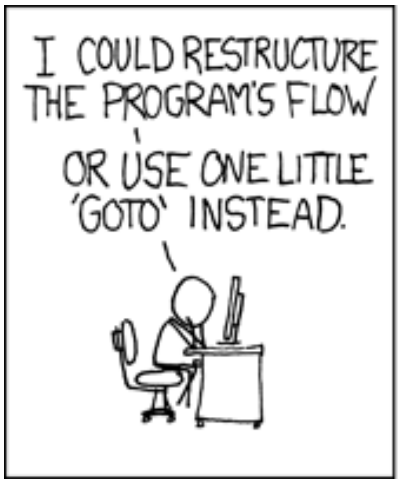
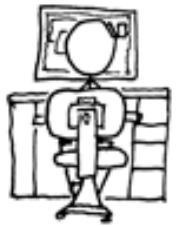
Admin

⭐ *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 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?

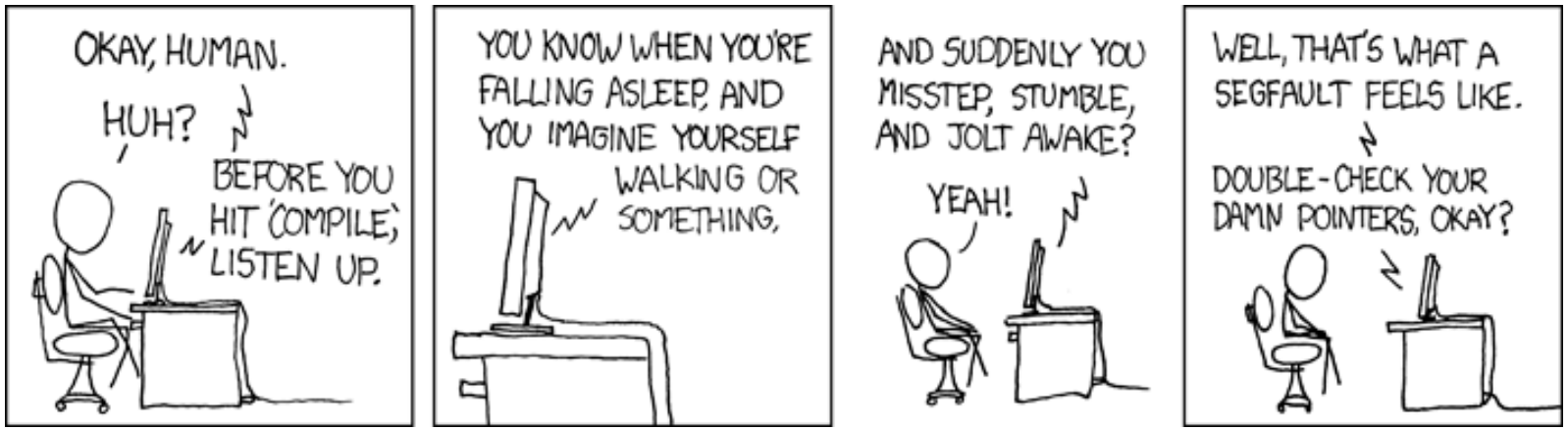
- Type system adds readability, some safety
- Pointee and level of indirection now explicit in the type
- Organize related data into contiguous locations, access using offset arithmetic

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)

When coding directly in assembly, you get what you see.

For C source, you may need to look at what compiler has generated to be sure of what you're getting.

What transformations are *legal* ?
What transformations are *desirable* ?

When Your C Compiler Is Too Smart For Its Own Good

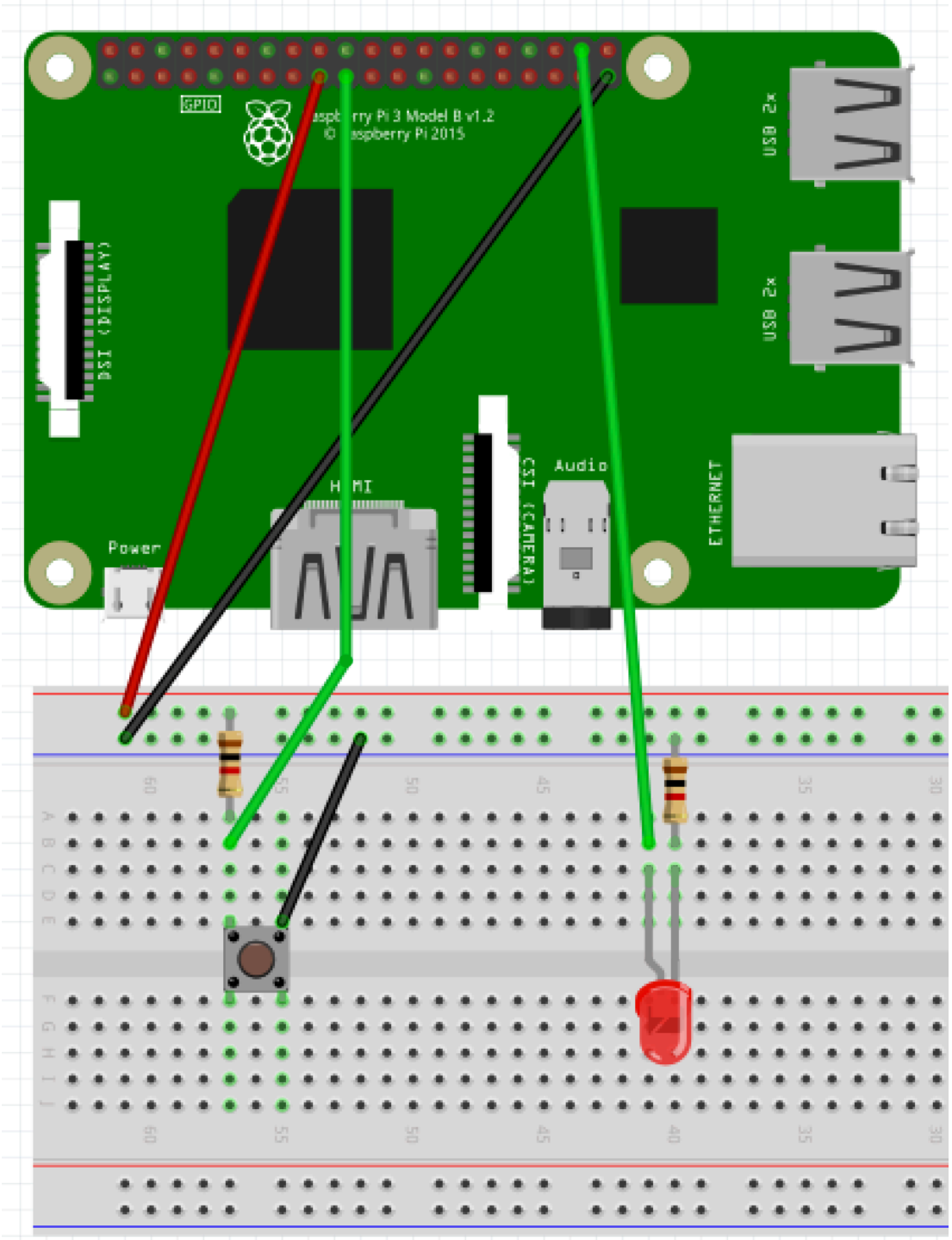
(or, why every systems programmer should be able to read assembly)

```
int i, j;

i = 1;
i = 2;
j = i;

// can be optimized to

i = 2;
j = i;

// is this ever not equivalent/ok?
```

button.c

The little button that wouldn’t

button.c: The little button that wouldn't



Want cool diagrams like this?
Check out fritzing.org

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. To read the current value, access the LEV (level) register. So writing to SET can change the value of LEV, a different memory address!

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

button.c: The little button that wouldn't

```
// This program waits until a button is pressed (GPIO 10)
// and turns on GPIO 20, then waits until the button is
//released and turns off GPI0 20

unsigned int * const FSEL1 = (unsigned int *)0x20200004;
unsigned int * const FSEL2 = (unsigned int *)0x20200008;
unsigned int * const SET0  = (unsigned int *)0x2020001C;
unsigned int * const CLR0  = (unsigned int *)0x20200028;
unsigned int * const LEV0  = (unsigned int *)0x20200034;


void main(void)
{
    *FSEL1 = 0; // configure GPIO 10 as input
    *FSEL2 = 1; // configure GPIO 20 as output

    while (1) {

        // wait until GPIO 10 is low (button press)
        while ((*LEV0 & (1 << 10)) != 0) ;

        // set GPIO 20 high
        *SET0 = 1 << 20;

        // wait until GPIO 10 is high (button release)
        while ((*LEV0 & (1 << 10)) == 0) ;

        // clear GPIO 20
        *CLR0 = 1 << 20;
    }
}
```

button.c: The little button that wouldn't

```
// This program waits until a button is pressed (GPIO 10)
// and turns on GPIO 20, then waits until the button is
//released and turns off GPI0 20

unsigned int * const FSEL1 = (unsigned int *)0x20200004;
unsigned int * const FSEL2 = (unsigned int *)0x20200008;
unsigned int * const SET0  = (unsigned int *)0x2020001C;
unsigned int * const CLR0  = (unsigned int *)0x20200028;
unsigned int * const LEV0  = (unsigned int *)0x20200034;


void main(void)
{
    *FSEL1 = 0; // configure GPIO 10 as input
    *FSEL2 = 1; // configure GPIO 20 as output

    while (1) {

        // wait until GPIO 10 is low (button press)
        while ((*LEV0 & (1 << 10)) != 0) ;

        // set GPIO 20 high
        *SET0 = 1 << 20;

        // wait until GPIO 10 is high (button release)
        while ((*LEV0 & (1 << 10)) == 0) ;

        // clear GPIO 20
        *CLR0 = 1 << 20;
    }
}
```

Compiling with –O2:

```
Disassembly of section .text.startup:

00000000 <main>:
   0:     ldr  r3, [pc, #28]  ; 24 <main+0x24>
   4:     ldr  r0, [r3, #52]  ; 0x34
   8:     mov  r1, #0
   c:     mov  r2, #1
  10:     tst  r0, #1024 ; 0x400
  14:     stmib     r3, {r1, r2}
  18:     bne  20 <main+0x20>
  1c:     b    1c <main+0x1c>
  20:     b    20 <main+0x20>
  24:     .word     0x20200000
```

button.c: The little button that wouldn't

```
// This program waits until a button is pressed (GPIO 10)
// and turns on GPIO 20, then waits until the button is
//released and turns off GPIO 20

unsigned int * const FSEL1 = (unsigned int *)0x20200004;
unsigned int * const FSEL2 = (unsigned int *)0x20200008;
unsigned int * const SET0  = (unsigned int *)0x2020001C;
unsigned int * const CLR0  = (unsigned int *)0x20200028;
unsigned int * const LEV0  = (unsigned int *)0x20200034;


void main(void)
{
    *FSEL1 = 0; // configure GPIO 10 as input
    *FSEL2 = 1; // configure GPIO 20 as output

    while (1) {

        // wait until GPIO 10 is low (button press)
        while ((*LEV0 & (1 << 10)) != 0) ;

        // set GPIO 20 high
        *SET0 = 1 << 20;

        // wait until GPIO 10 is high (button release)
        while ((*LEV0 & (1 << 10)) == 0) ;

        // clear GPIO 20
        *CLR0 = 1 << 20;
    }
}
```

Compiling with –O2:

```
Disassembly of section .text.startup:

00000000 <main>:
   0:   ldr  r3, [pc, #28]  ; 24 <main+0x24>
   4:   ldr  r0, [r3, #52]  ; 0x34
   8:   mov  r1, #0
   c:   mov  r2, #1
  10:   tst  r0, #1024 ; 0x400
  14:   stmib     r3, {r1, r2}
  18:   bne  20 <main+0x20>
  1c:   b    1c <main+0x1c>   ?
  20:   b    20 <main+0x20>
  24:   .word     0x20200000  ?
```

button.c: The little button that wouldn't

```
// This program waits until a button is pressed (GPIO 10)
// and turns on GPIO 20, then waits until the button is
//released and turns off GPIO 20

unsigned int * const FSEL1 = (unsigned int *)0x20200004;
unsigned int * const FSEL2 = (unsigned int *)0x20200008;
unsigned int * const SET0  = (unsigned int *)0x2020001C;
unsigned int * const CLR0  = (unsigned int *)0x20200028;
unsigned int * const LEV0  = (unsigned int *)0x20200034;


void main(void)
{
    *FSEL1 = 0; // configure GPIO 10 as input
    *FSEL2 = 1; // configure GPIO 20 as output

    while (1) {

        // wait until GPIO 10 is low (button press)
        while ((*LEV0 & (1 << 10)) != 0) ;

        // set GPIO 20 high
        *SET0 = 1 << 20;

        // wait until GPIO 10 is high (button release)
        while ((*LEV0 & (1 << 10)) == 0) ;

        // clear GPIO 20
        *CLR0 = 1 << 20;
    }
}
```

Compiling with –O2:

```
Disassembly of section .text.startup:

00000000 <main>:
   0:     ldr  r3, [pc, #28]  ; 24 <main+0x24>
   4:     ldr  r0, [r3, #52]  ; 0x34
   8:     mov  r1, #0
   c:     mov  r2, #1
  10:     tst  r0, #1024 ; 0x400
  14:     stmib     r3, {r1, r2}
  18:     bne  20 <main+0x20>
  1c:     b    1c <main+0x1c>
  20:     b    20 <main+0x20>
  24:     .word     0x20200000
```

What happened to our testing loops??

`volatile`

For an ordinary variable, the compiler can use its knowledge of when it is read/written to optimize accesses as long as it keeps the same externally visible behavior.

However, for a variable that can be read/written externally (by another process, by peripheral), these optimizations will not be valid.

The **volatile** qualifier applied to a variable informs the compiler that it cannot remove, coalesce, cache, or reorder references. The generated assembly must faithfully execute each access to the variable as given in the C code.

button.c: The little button that could

Because we have GPIO pins on the Raspberry Pi, we need to give hints to the C compiler to not optimize out pin reads — they can change externally to the program!

So, we use the `volatile` keyword in front of hardware addresses to do this:

```
volatile unsigned int * const FSEL1 = (unsigned int *)0x20200004;
volatile unsigned int * const FSEL2 = (unsigned int *)0x20200008;
volatile unsigned int * const SET0  = (unsigned int *)0x2020001C;
volatile unsigned int * const CLR0  = (unsigned int *)0x20200028;
volatile unsigned int * const LEV0  = (unsigned int *)0x20200034;
```

button.c: The little button that could

There are other times to use volatile, too — delays have a similar problem:

```
#define DELAY 500000000

int main()
{
    for (int i=0; i < DELAY; i++);

    return 0;
}
```

```
$ objdump -d testLoop.o

testLoop.o:     file format elf32-littlearm

Disassembly of section .text.startup:

00000000 <main>:
   0:  e3a00000  mov r0, #0
   4:  e12fff1e  bx  lr
```

button.c: The little button that could

There are other times to use volatile, too — delays have a similar problem:

```
#define DELAY 500000000

int main()
{
    for (int i=0; i < DELAY; i++);

    return 0;
}
```

```
$ objdump -d testLoop.o

testLoop.o:     file format elf32-littlearm

Disassembly of section .text.startup:

00000000 <main>:
   0:   e3a00000    mov r0, #0
   4:   e12fff1e    bx  lr
```

No loop — it has been optimized out!

button.c: The little button that could

There are other times to use volatile, too — delays have a similar problem:

```
#define DELAY 500000000

int main()
{
    for (volatile int i=0; i < DELAY; i++);

    return 0;
}
```

```
Disassembly of section .text.startup:

00000000 <main>:
   0: e24dd008    sub   sp, sp, #8
   4: e3a03000    mov   r3, #0
   8: e58d3004    str   r3, [sp, #4]
   c: e59d3004    ldr   r3, [sp, #4]
  10: e59f2028    ldr   r2, [pc, #40]     ; 40 <main+0x40>
  14: e1530002    cmp   r3, r2
  18: ca000005    bgt   34 <main+0x34>
  1c: e59d3004    ldr   r3, [sp, #4]
  20: e2833001    add   r3, r3, #1
  24: e58d3004    str   r3, [sp, #4]
  28: e59d3004    ldr   r3, [sp, #4]
  2c: e1530002    cmp   r3, r2
  30: dafffff9    ble   1c <main+0x1c>
  34: e3a00000    mov   r0, #0
  38: e28dd008    add   sp, sp, #8
  3c: e12fff1e    bx    lr
  40: 1dcd64ff    .word 0x1dcd64ff
```

The loop remains when we use volatile.

C Functions

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
```

ARM quirk: 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

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;
}
```

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***

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 a function needs scratch space for an intermediate value, which type of register should it choose?

2. What must a function do when it wants to use a caller-owed register?

3. What is the advantage in having some registers callee-owned and others caller-owned? Why not treat all same?

4. How can we implement nested calls when we only have a single shared lr register? A() -> B() -> C() -> …

The Stack to the Rescue

Program Stack

Region in memory to store local variables, scratch space, save register values

- **LIFO: push adds value on top of stack, pop removes lastmost value**
- **`r13` (alias `sp`) points to end of stack**
- **stack grows down**
  - **newer values at lower addresses**
  - **push subtracts from `sp`**
  - **pop adds to `sp`**
- **push/pop are aliases for a general instruction (load/store multiple with writeback)**

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

```
// main.c
void main(void)
{
    add10(3);
}

int add10(int a)
{
    int arr[100];
    return add(arr, 100);
}
```

Not to scale

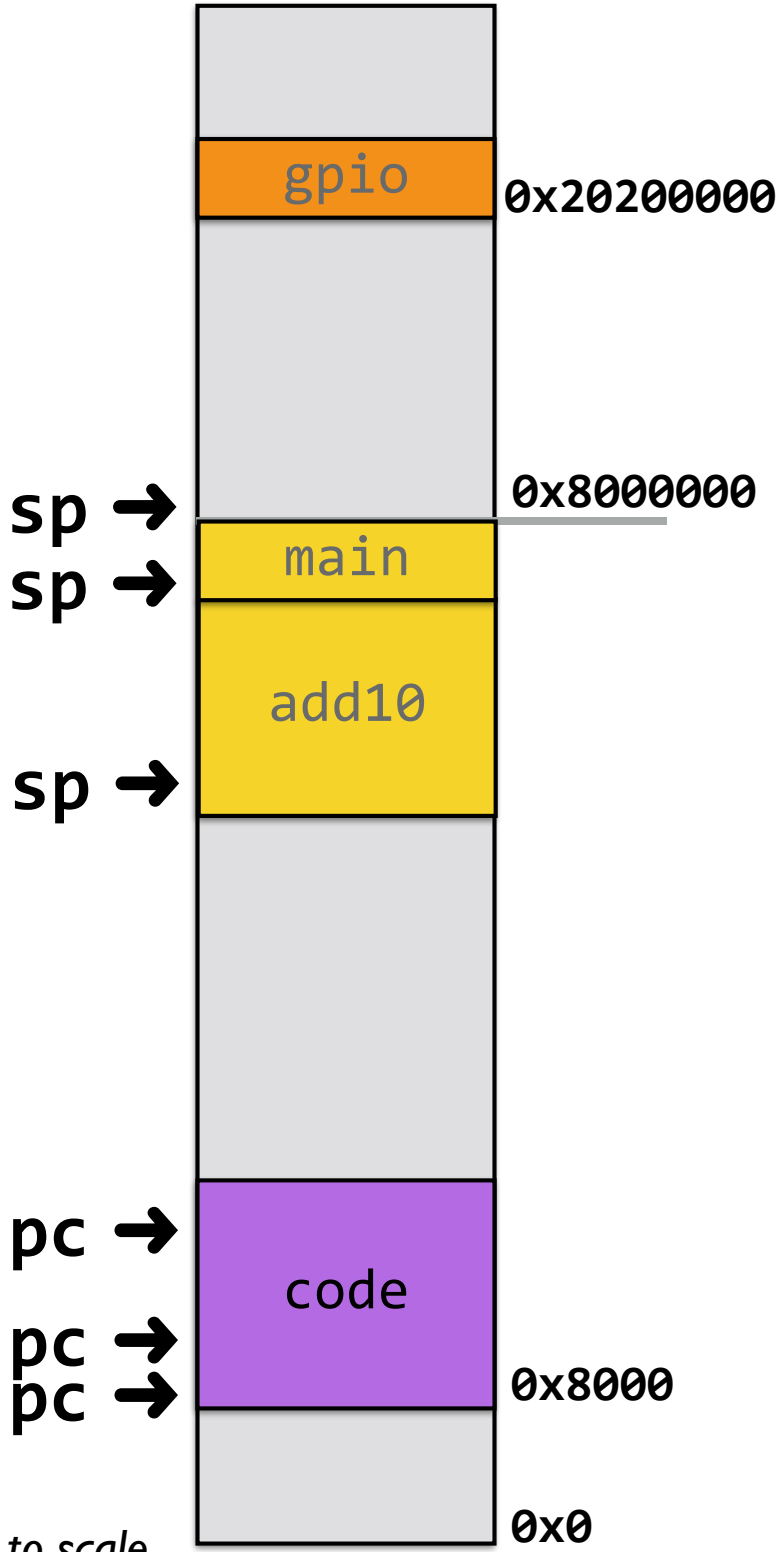
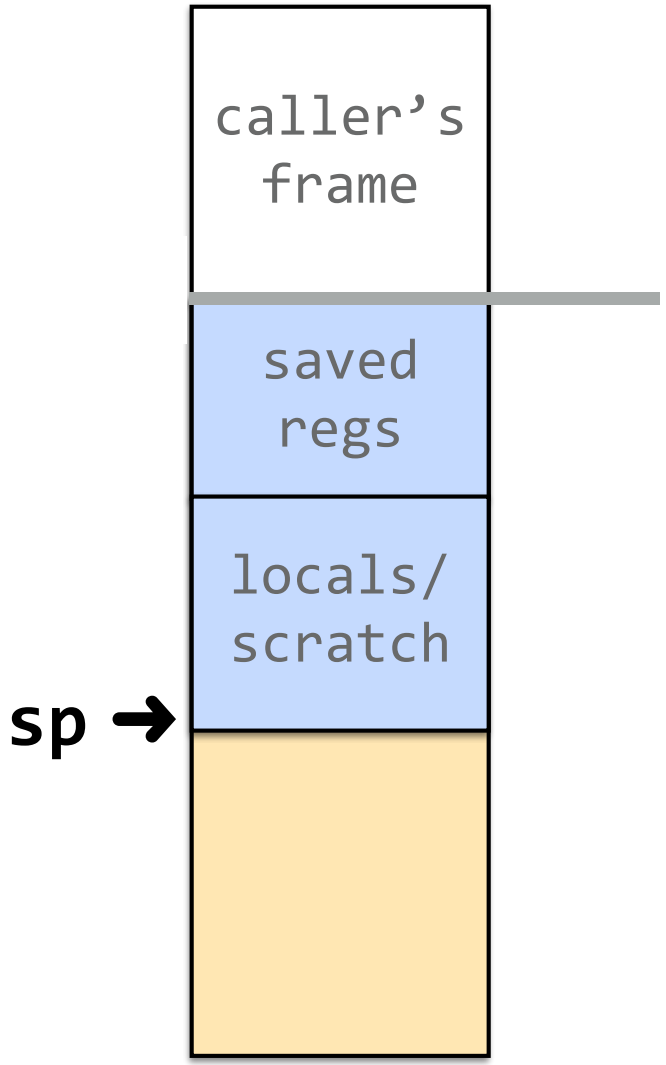


Diagram not to scale

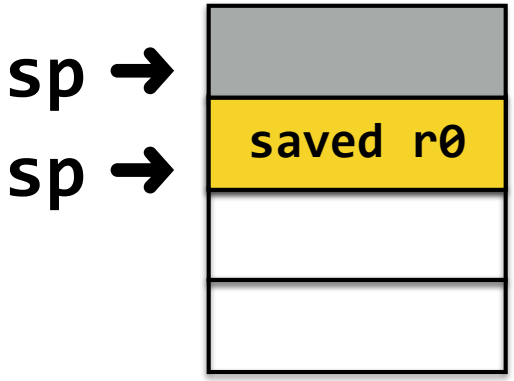
Single stack frame

```
int add10(int a, int b)
{
   int c = 2*a;
   ...
   return c;
}
```



Stack operations

```
// PUSH (store reg to stack)
// *–sp = r0
// decrement sp before store
push {r0}
// equivalent to:
        str r0, [sp, #-4]!
```



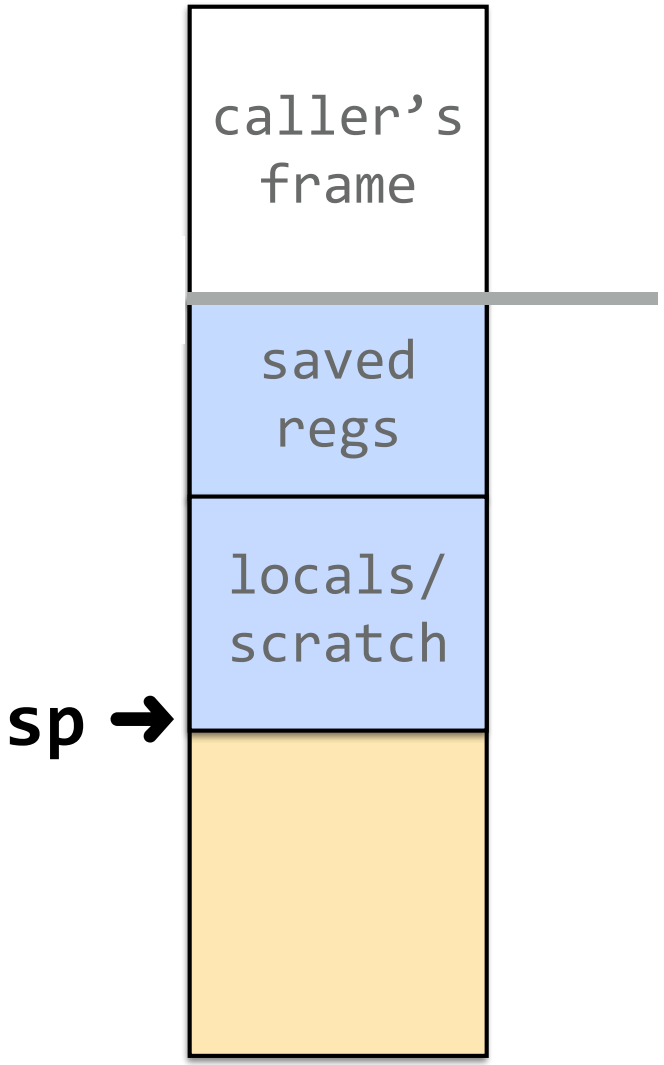

```
// POP (restore reg from stack)
// r0 = *sp++
// increment sp after load
pop {r0}
// equivalent to:
        ldr r0, [sp], #4
```

“Full Descending” stack

sp in constant motion

Access values on stack using sp-relative addressing, but ….

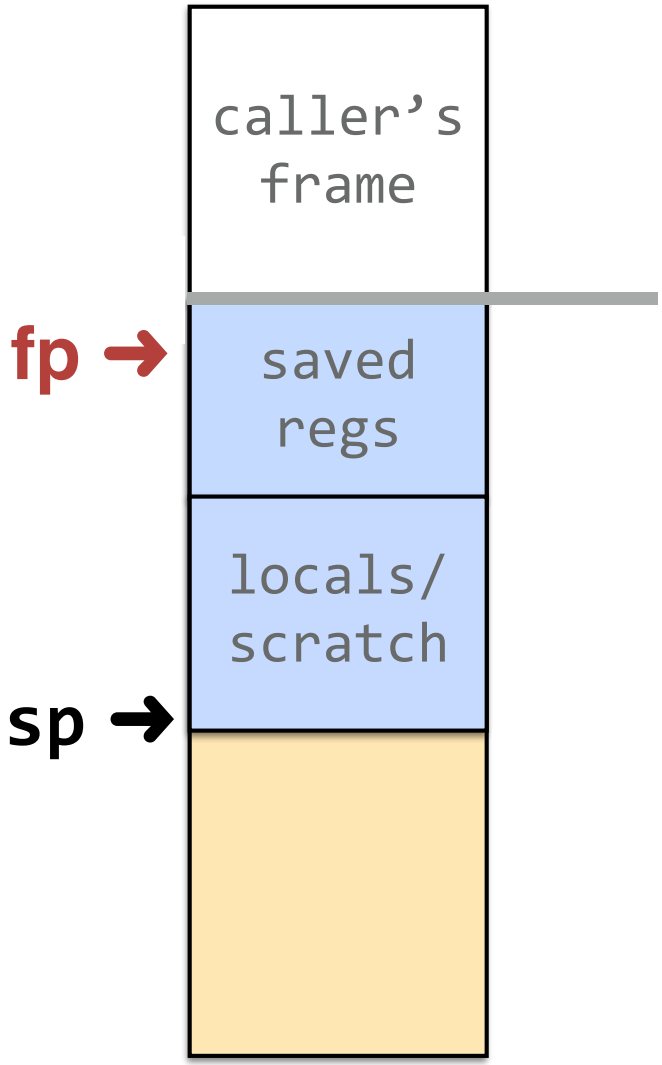
sp is constantly changing!
(push, pop, add sp, sub sp)



Add frame pointer (fp)

Dedicate `fp` register to be used as fixed anchor

Offsets relative to `fp` stay constant!



APCS “full frame”

APCS = ARM Procedure Call Standard

Conventions for use of frame pointer + frame layout that allows for reliable stack introspection

gcc CFLAGS to enable: `-mapcs-frame`

`r12` used as `fp`

Adds a prologue/epilogue to each function that sets up/ tears down the standard frame and manages fp

You'll use this in assignment 4

Trace APCS full frame

Prolog

push fp, sp*, lr, pc

set **fp** to first word of stack frame

Body

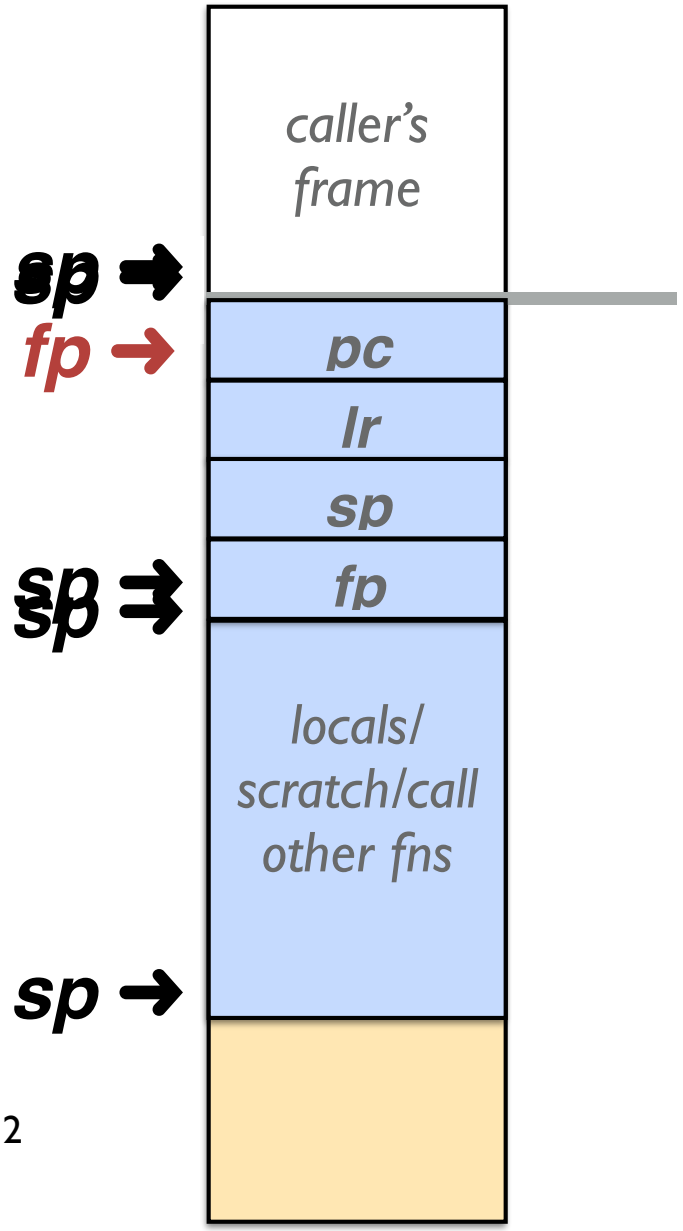
fp stays anchored

access data on stack **fp**-relative

offsets won't vary even if **sp** changing

Epilog

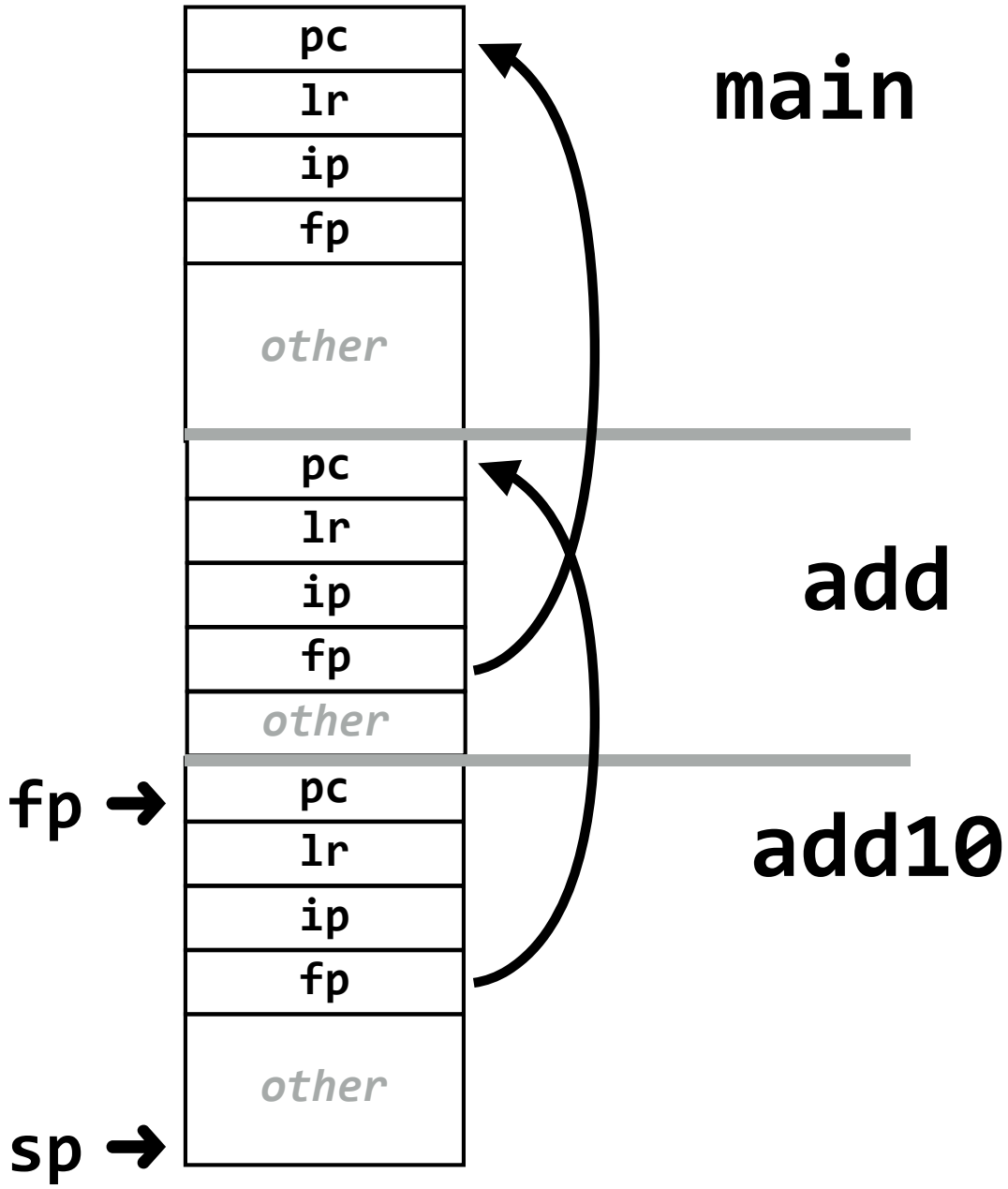
pop fp, sp*, lr, pc*



* I am fudging a bit about use of push and pop
The **sp** register cannot be directly pushed/popped, instead moved through r12
pc cannot be popped at end, is manually removed from stack

FPs form linked chain

other =
additional saved regs,
locals,
scratch



APCS Pros/Cons

+ Anchored fp, offsets are constant
+ Standardized frame layout enables introspection
+ Backtrace for debugging
+ Unwind stack on exception

- Expensive, every function call affected
  - prolog/epilog add ~5 instructions
  - 4 registers push/pop => add 16 bytes per frame
  - consumes one of our precious registers

```
// start.s

// Need to initialize fp = NULL
// to terminate end of chain

    mov sp, #0x8000000
    mov fp, #0    // fp = NULL
    bl main
```

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/