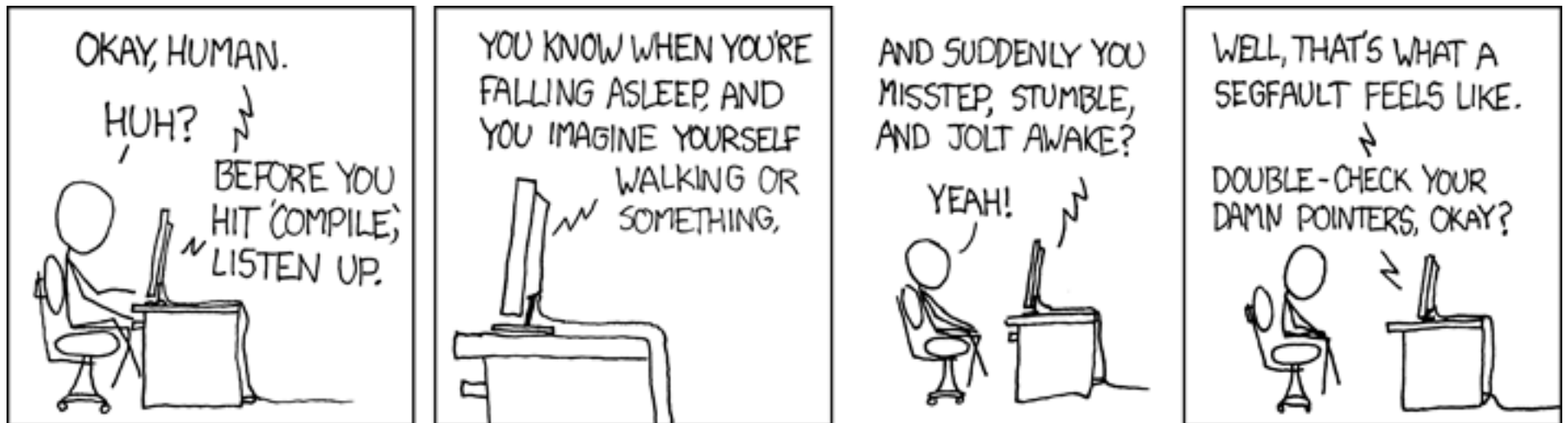


Goals for today

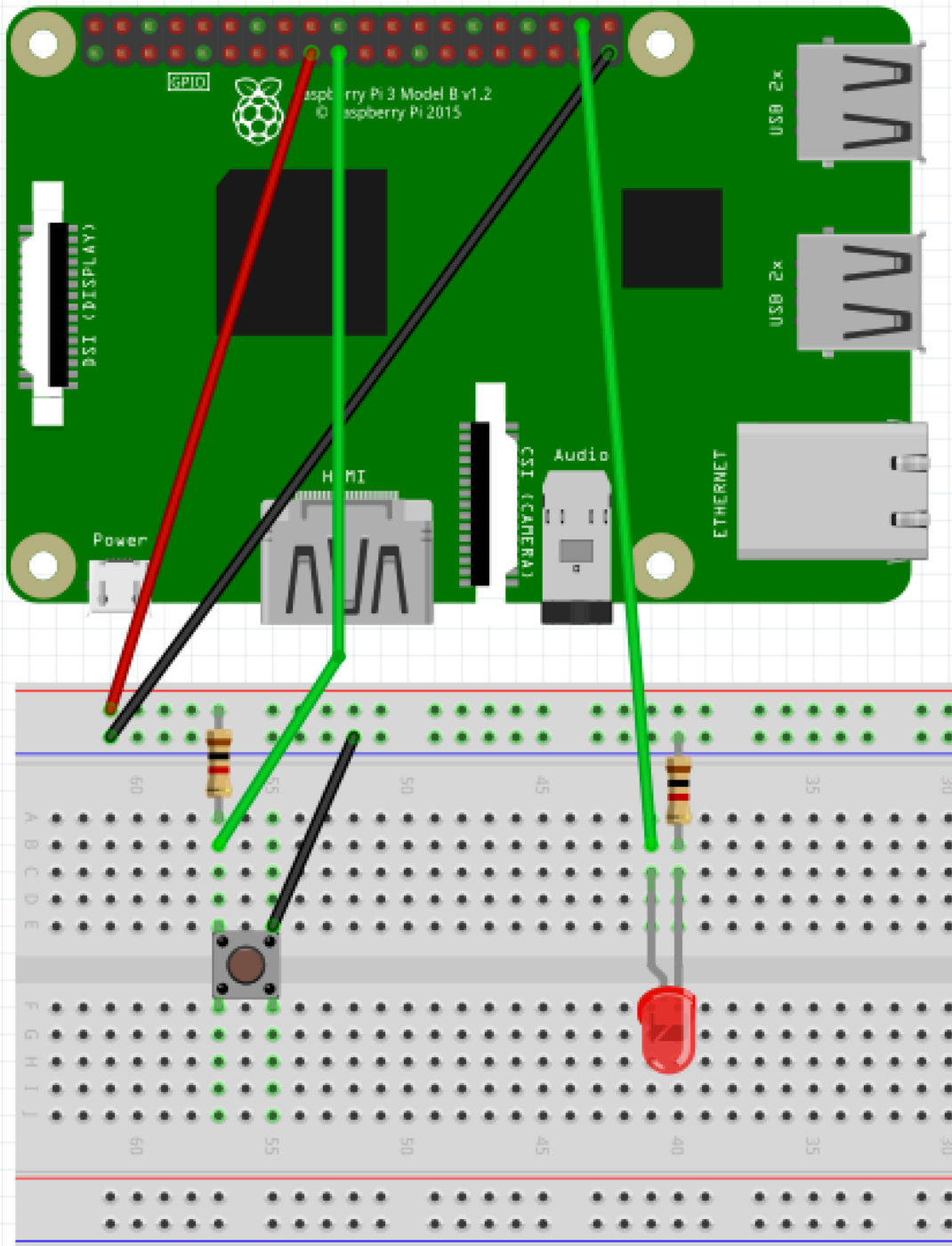
- The little button that wouldn't :(
 - the `volatile` keyword
- How do functions work in C?
- Management of runtime stack, register use
- Multiple loads and stores: `ldm` and `stm`



button.c

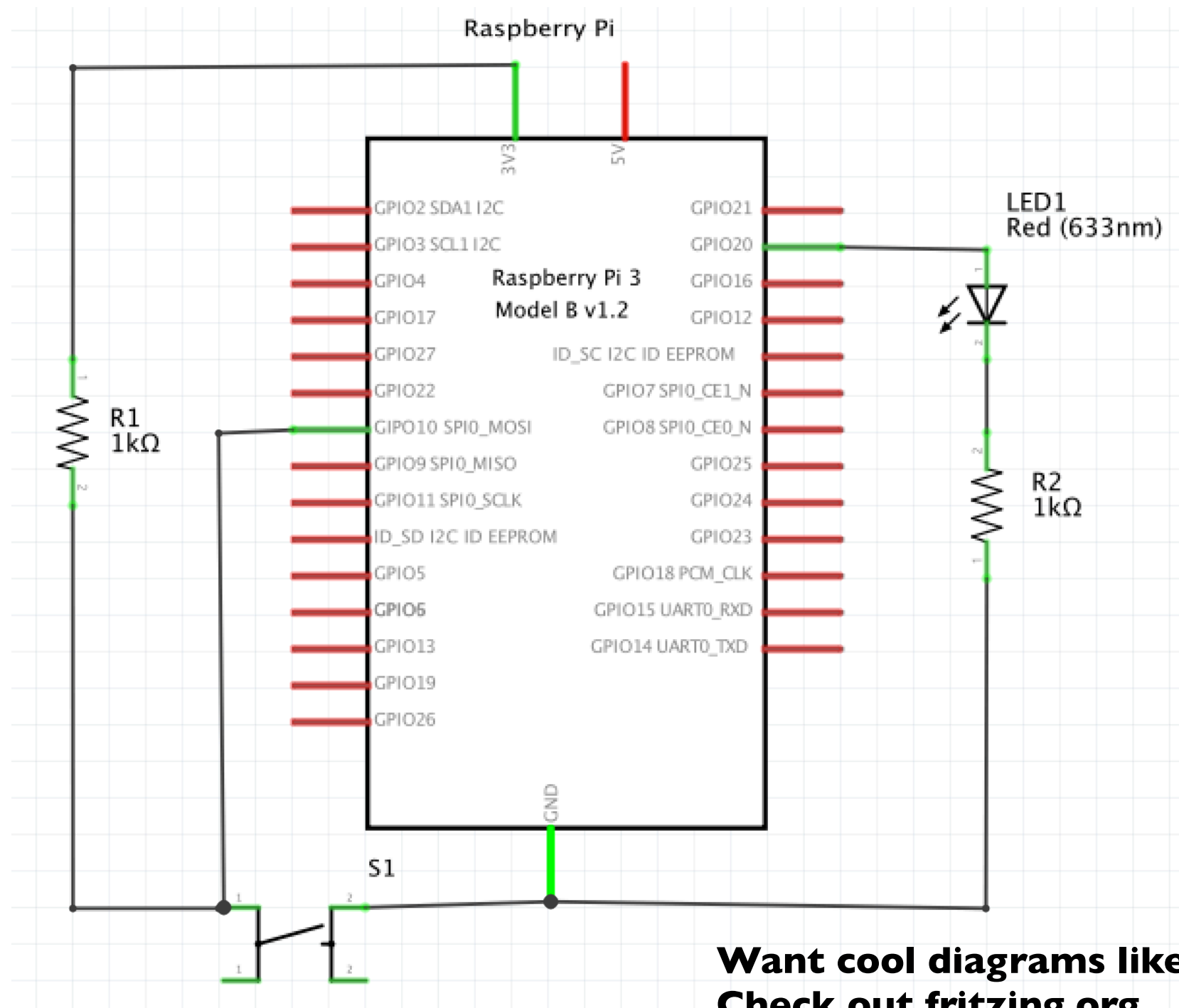
The little button that wouldn't

button.c: The little button that wouldn't



**Want cool diagrams like this?
Check out fritzing.org**

button.c: The little button that wouldn't



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 CLRO  = (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
        *CLRO = 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 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 CLRO  = (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
        *CLRO = 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 CLRO  = (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
```

```
        *CLRO = 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 CLRO  = (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
        *CLRO = 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??

Peripheral Registers

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

These registers may behave differently than memory.

For example: Writing a 1 into a bit in a SET register causes 1 to be output; writing a 0 into a bit in SET register does not affect the output value. Writing a 1 to the CLR register, sets the output to 0; write a 0 to a clear register has no effect. Neither SET or CLR can be read. To read the current value use the LEV (level) register.

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: daffffff9    ble    1c <main+0x1c>
34: e3a00000    mov    r0, #0
38: e28dd008    add    sp, sp, #8
3c: e12ffff1e    bx     lr
40: 1dcd64fff    .word 0x1dcd64fff
```

The loop remains when we use volatile.

What is ‘bare metal’?

The default build process for C assumes a *hosted* environment. It provides standard libraries, all the stuff that happens before `main`.

To build bare-metal, our makefile disables these defaults; we must supply our own versions when needed.

```
int getRandomNumber()
{
    return 4; // chosen by fair dice roll.
              // guaranteed to be random.
}
```

Makefile settings

Compile freestanding

```
CFLAGS = -ffreestanding
```

Link without standard libs and start files

```
LDFLAGS = -nostdlib
```

Link with gcc to support division

```
LDLIBS = -lgcc
```

Must supply own replacement for libs/start

That's where the fun is...!

C++ source #1 ×

A ▾



```
1 void wipe1(int arr[])
2 {
3     arr[1] = 0;
4 }
5
6 struct point {
7     int x, y, z;
8 };
9
10 void wipe2(struct point *ptr)
11 {
12     ptr->y = 0;
13 }
```

ARM gcc 5.4 (Editor #1, Compiler #1) ×

ARM gcc 5.4 ▾

-Og -ffree-standing -marm

11010

.LX0:

.text

//

Intel

A ▾



```
1 wipe1(int*):
2     mov     r3, #0
3     str     r3, [r0, #4]
4     bx      lr
5 wipe2(point*):
6     mov     r3, #0
7     str     r3, [r0, #4]
8     bx      lr
9
```

loop:

```
ldr r0, =0x2020001C    // set pin
str r1, [r0]
```

```
mov r2, #0x3F0000      // delay loop
```

wait1:

```
subs r2, #1
bne wait1
```

```
ldr r0, =0x20200028    // clear pin
str r1, [r0]
```

```
mov r2, #0x3F0000      // delay loop
```

wait2:

```
subs r2, #1
bne wait2
```

b loop

```
[8000]    ldr r0, =0x2020001C
[8004]    str r1, [r0]
[8008]    b delay
[800c]    ldr r0, =0x20200028
[8010]    str r1, [r0]
[8014]    b delay
[8018]    b loop
```

delay:

```
[801c]    mov r2, #0x3F0000
[8020]    wait:
[8024]        subs r2, #1
[8028]        bne wait
// but...  where to go next?
```

```
ldr r0, =0x2020001C
str r1, [r0]
mov r14, pc
b delay
ldr r0, =0x20200028
str r1, [r0]
mov r14, pc
b delay
b loop
```

```
delay:
mov r2, #0x3F0000
wait:
    subs r2, #1
    bne wait
mov pc, r14
```

To return from a function, we need to know what pc we called it from, so we can continue after.

```
ldr r0, =0x2020001C
str r1, [r0]
mov r0, #0x3F0000
mov r14, pc
b delay
ldr r0, =0x20200028
str r1, [r0]
mov r0, #0x3F0000 >> 2
mov r14, pc
b delay
b loop
```

delay:

```
subs r0, #1
```

wait:

```
bne wait
```

```
mov pc, r14
```

We also need to pass parameters into functions.

Anatomy of C function call

```
int sum(int n)
{
    int total = 0;
    for (int i = 1; i < n; i++)
        total += i;
    return total;
}
```

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 is ARM embedded ABI
(“none” refers to no hosting OS)

Mechanics of call/return

Caller puts up to 4 arguments in r0-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 mnemonic for r14

Caller and Callee

caller - function doing the calling

callee - function called

main is caller of add10

add10 is callee of main

+ caller of add

```
void main(void) {  
    add10(3);  
}
```

```
void add10(int a) {  
    add(10, a);  
}
```

```
int add(int x, int y) {  
    return x + y;  
}
```

Register Ownership

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

- **Callee** can change these registers
- **Caller** cedes to callee, cannot assume value will be preserved across call to callee

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

- **Callee** must preserve values in these registers
- **Caller** retains ownership, expects value to be same after call as it was before call

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-owned 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, #0x80000000
bl main
```

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

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

Not to scale

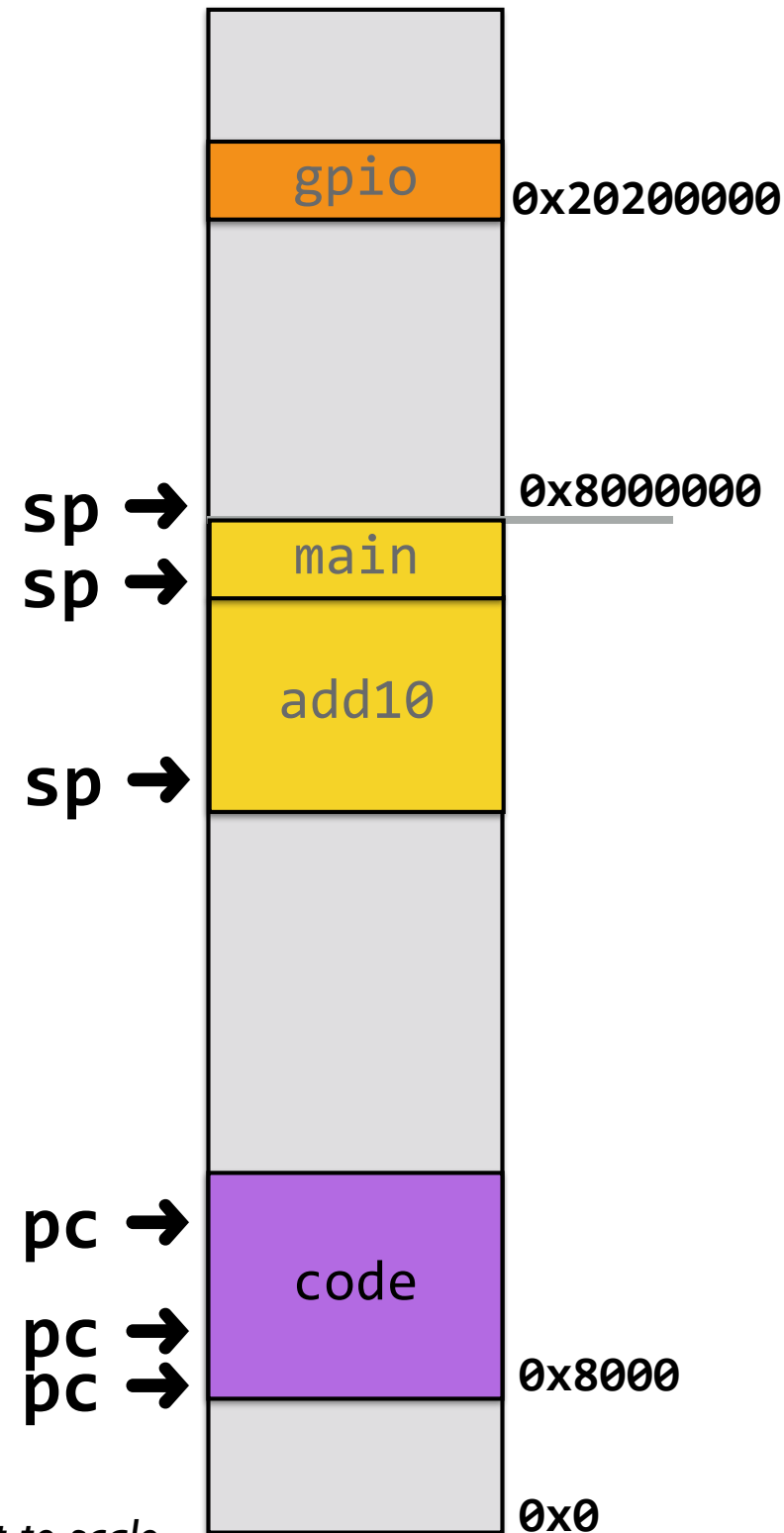
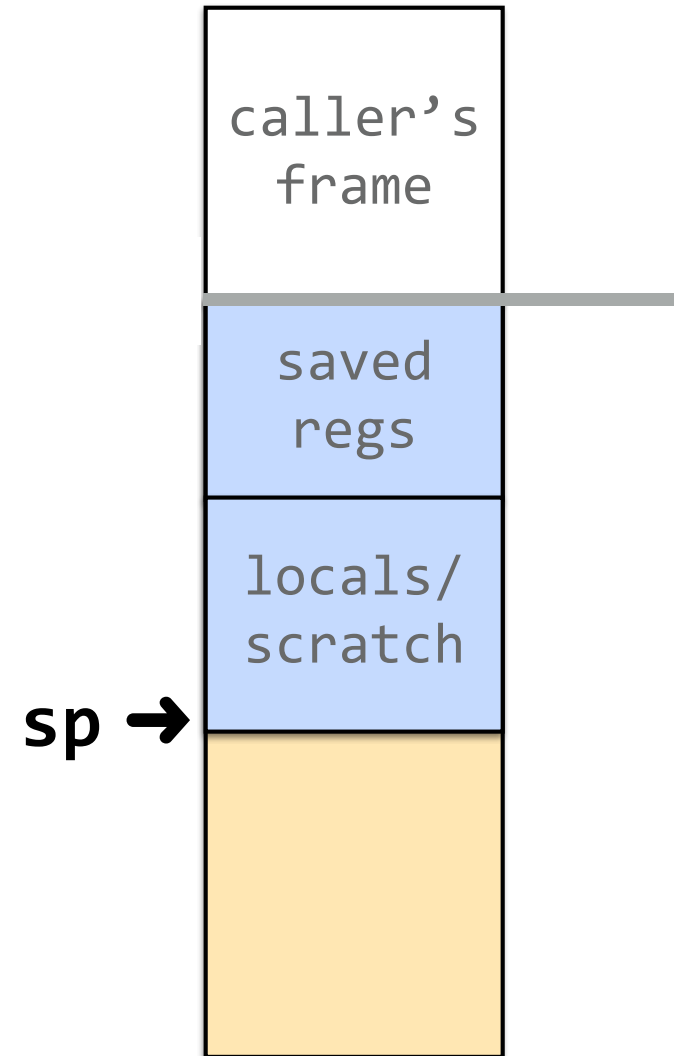


Diagram not to scale

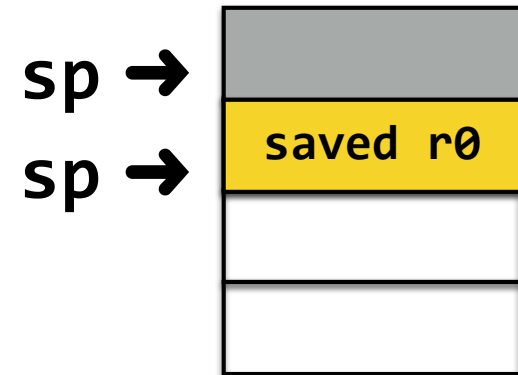
Single stack frame

```
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


```
int f(int a, int b)
{
    int c = g(a);
    return b + c;
}
```

If f calls g...

Why do they collide on use of `lr` ?

Is there similar collision for `r0`? `r1`?

What do we do about it?

use stack as temp storage!

example.c

r0
r1
r2
r3
lr
sp
pc

0x80000000
0x7fffffff
0x7fffffff8
0x7fffffff4
0x7ffffff0
0x7ffffec
0x7ffffe8
0x7ffffe4
0x7ffffe0

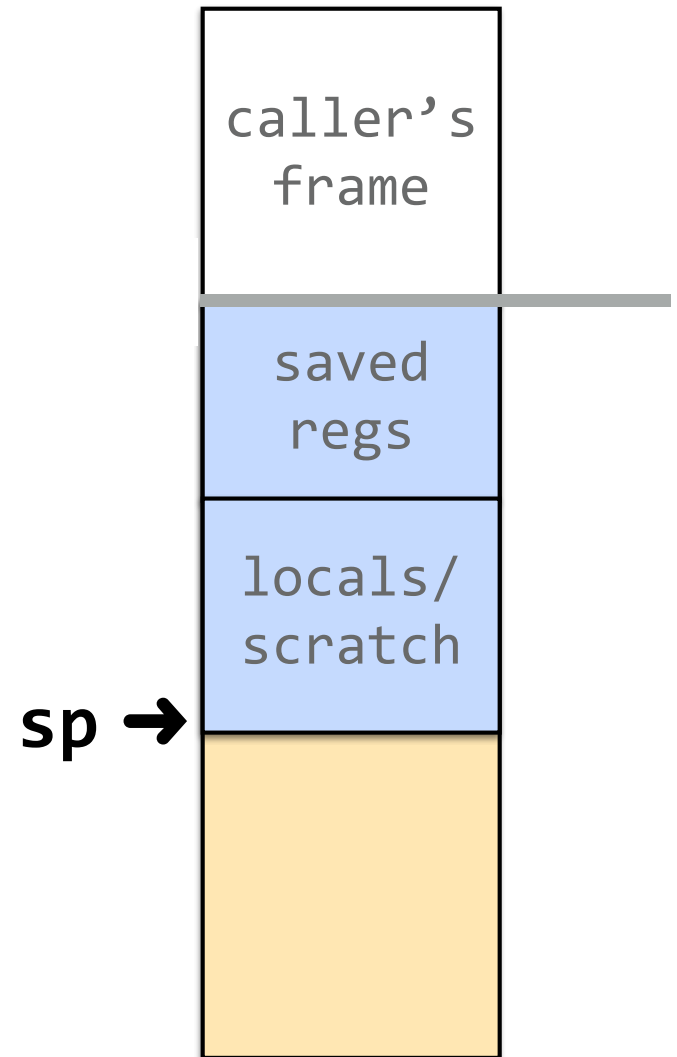


	0x80000000	
	0x7fffffff	
	0x7ffffff8	
	0x7ffffff4	
	0x7ffffff0	
	0x7fffffec	
	0x7fffffe8	
	0x7fffffe4	
r0	0x7fffffe0	
r1	:	:
r2	0x7ffffe70	
r3	0x7ffffe6c	
lr	0x7ffffe68	
sp	0x7ffffe64	
pc	0x7ffffe60	
	0x7ffffe5c	
	0x7ffffe58	
	0x7ffffe54	

sp in constant motion

Access values on stack using
sp-relative addressing, but

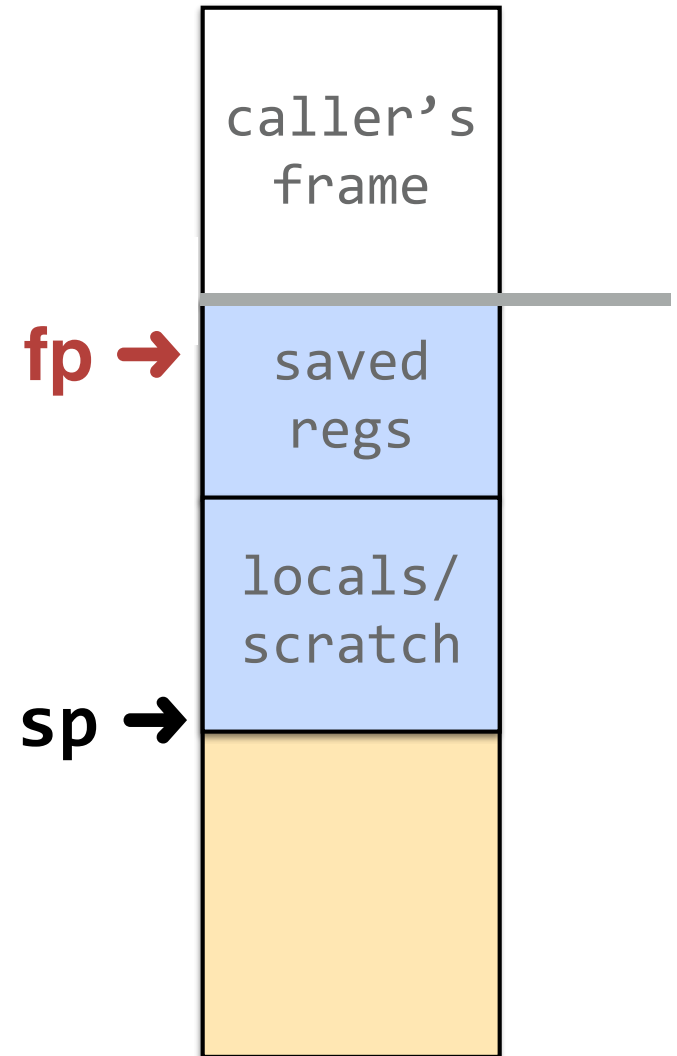
sp is constantly changing!
(push, pop, add sp, sub sp)



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

Prolog

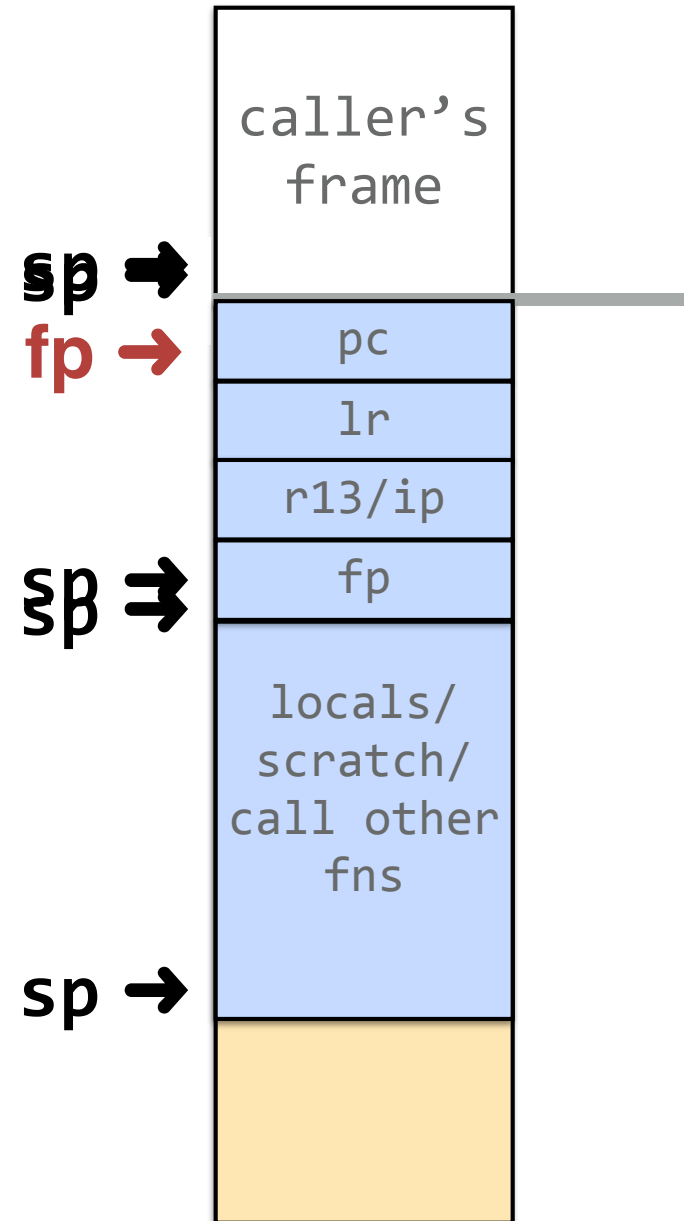
stmfd fp, sp, lr, pc
set fp to first word of stack frame

Body

fp stays anchored
access data on stack fp-relative
offsets won't vary even if sp changing

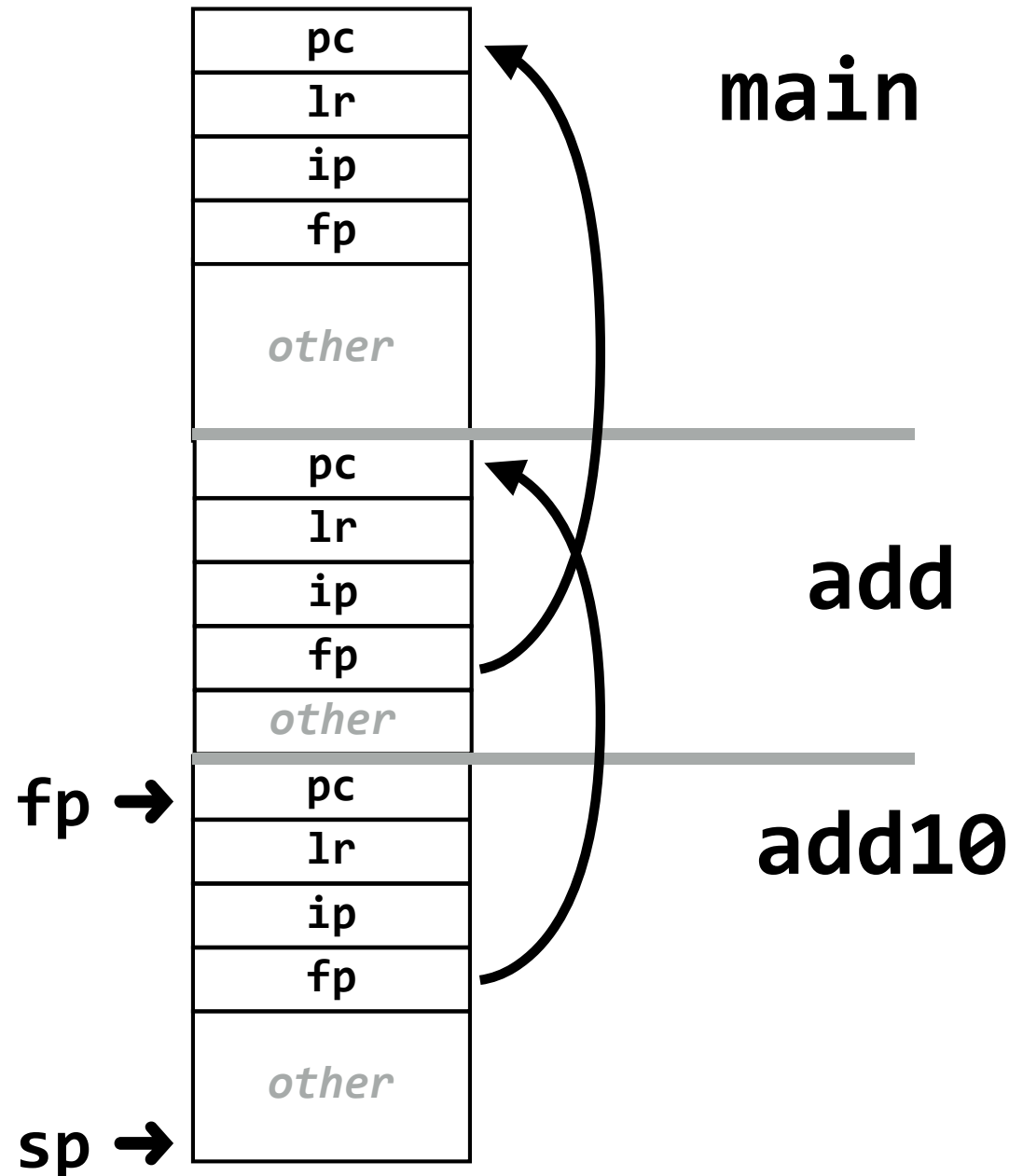
Epilog

ldmia fp, sp, lr
can't pop pc (**why not?**), manually adjust stack



FPS form linked chain

other =
additional saved regs,
locals,
scratch



```
// start.s
```

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

```
mov sp, #0x8000000
```

```
mov fp, #0      // fp = NULL
```

```
bl main
```

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