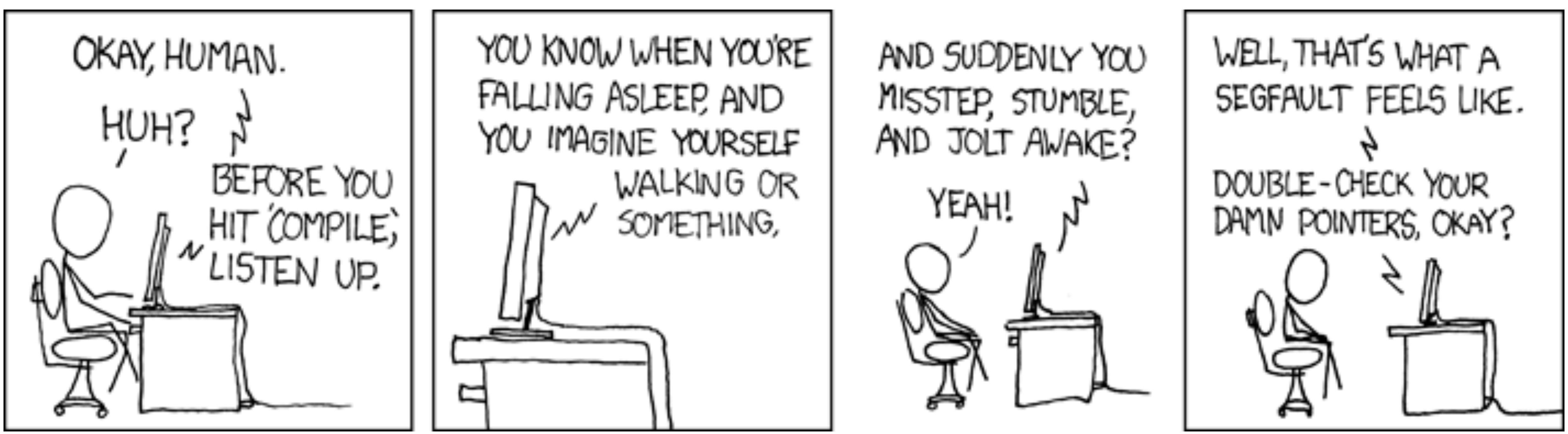
Goals for today

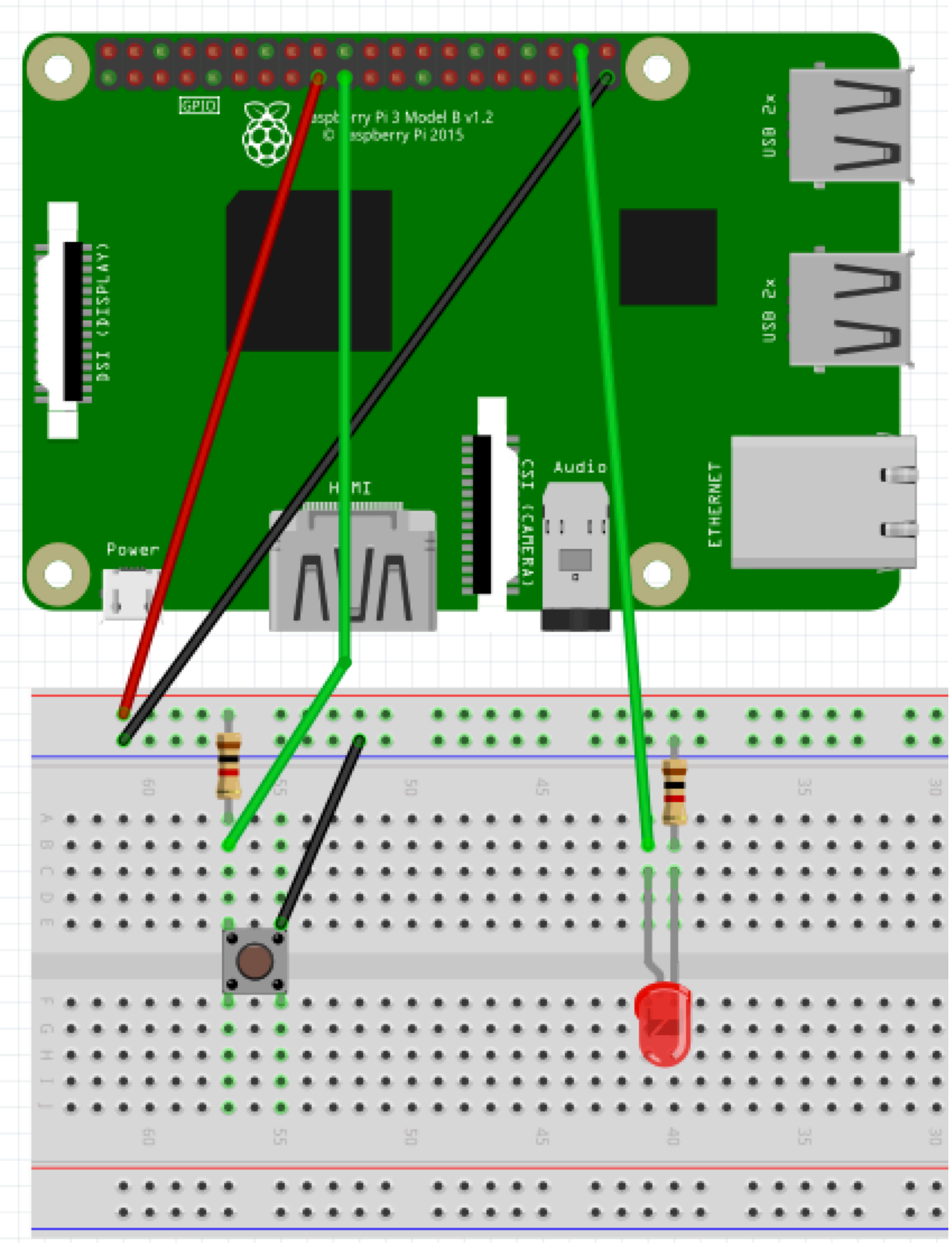
- The little button that wouldn't :(
  - the `volatile` keyword
- Pointer operations => ARM addressing modes
- Implementation of C function calls
- Management of runtime stack, register use



button.c

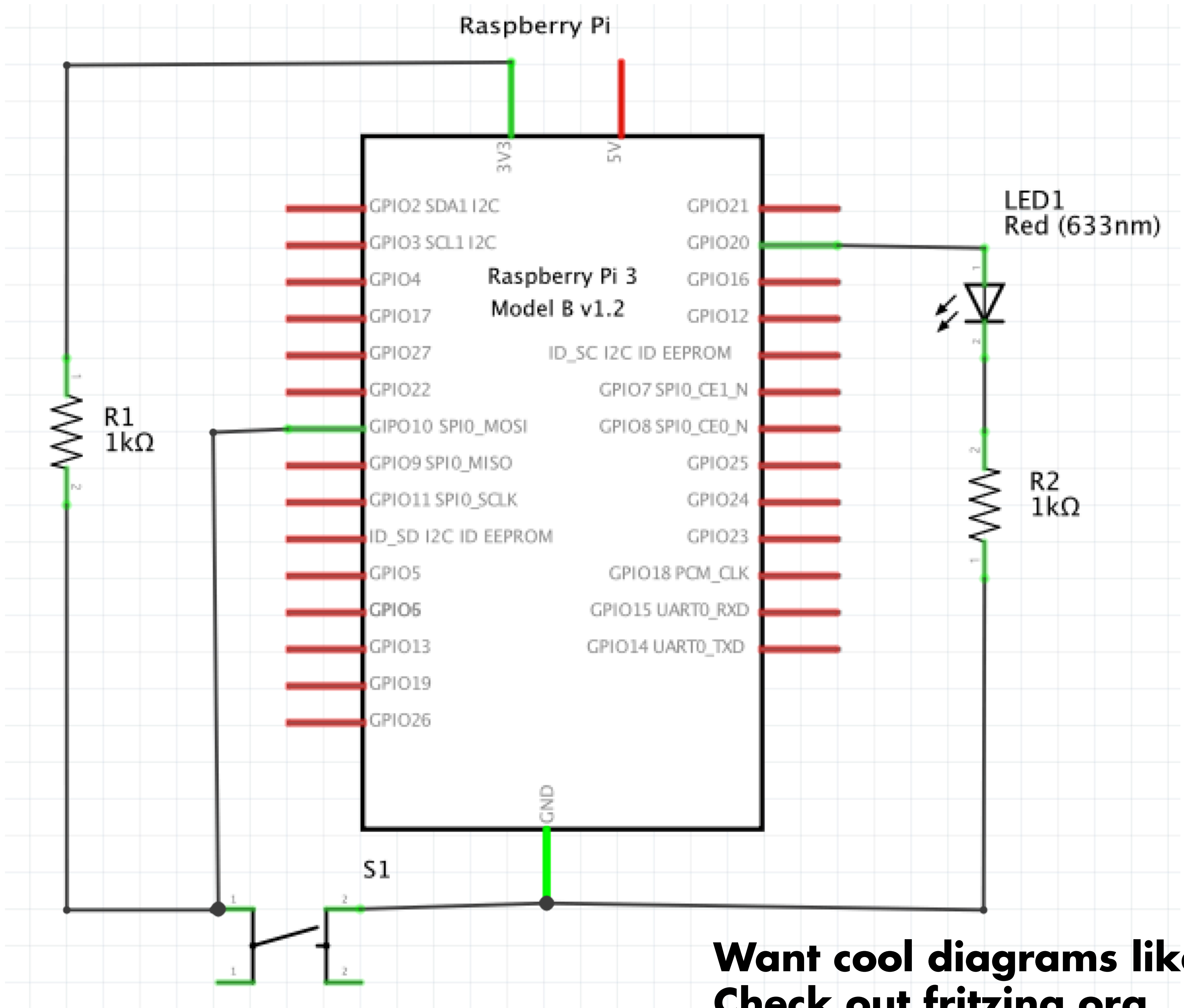
The little button that wouldn’t

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



Want cool diagrams like this?
Check out fritzing.org

button.c: The little button that wouldn’t

```c
// 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;
    }
}
```

button.c: The little button that wouldn’t

```c
// 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

```c
// 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??

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;
}
```

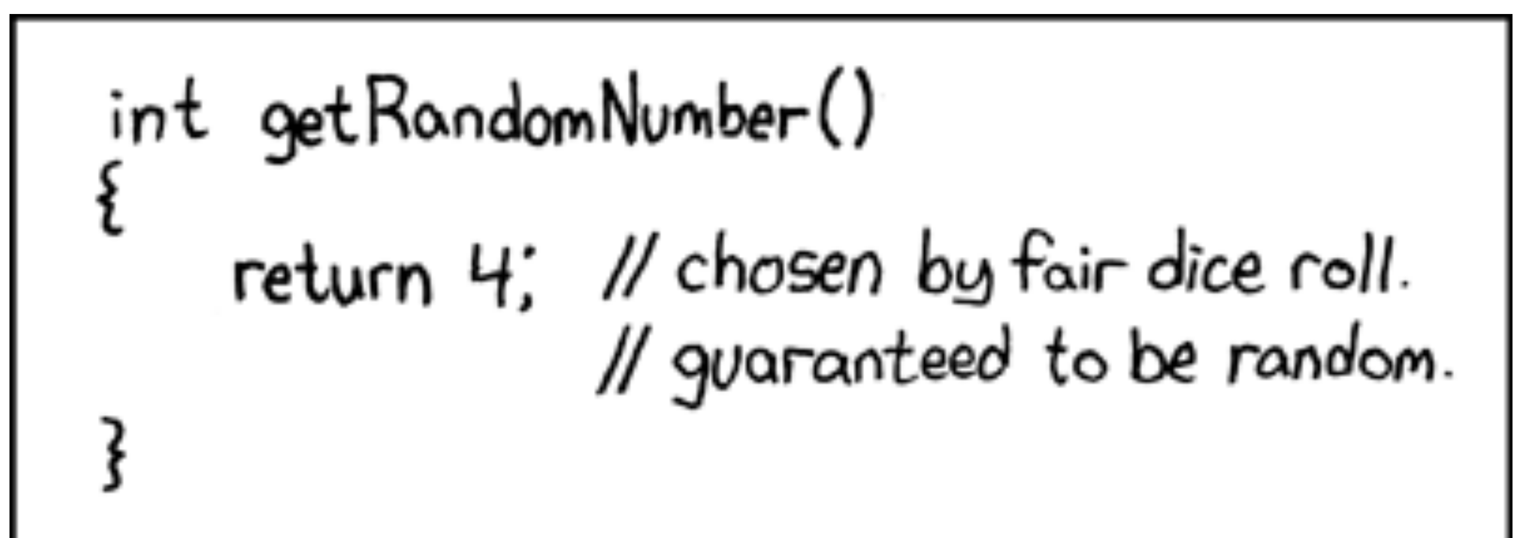
```
Disassembly of section .text.startup:

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: daffff9     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.

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.



Makefile settings

Compile freestanding

```
CFLAGS =-ffreestanding
```

Link without standard libs and start files

```
LDFLAGS = -nostdlib
```

Link with gcc to support division (violates

```
LDLIBS = -lgcc
```

Must supply own replacement for libs/start

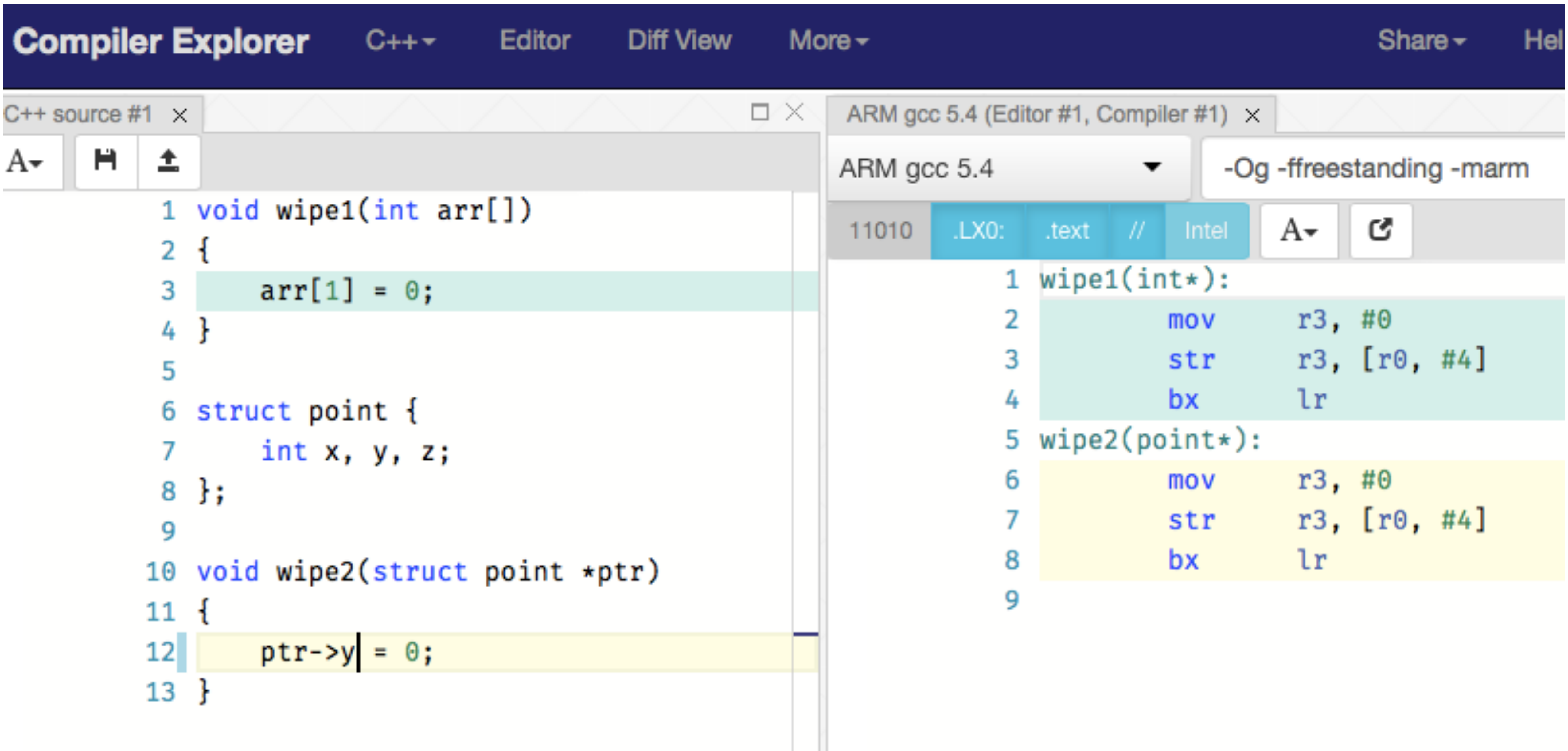
That's where the fun is…!

Pointers: more gain than pain!

"The fault, dear Brutus, is not in our stars
But in ourselves, that we are underlings."
Julius Caesar (I, ii, 140-141)

Refer to data by address or relative position is very useful!

- Sharing instead of copying
- Access to fields of a struct
- Array elements accessed by index
- Construct linked structures (lists, trees, graphs)


Compiler Explorer
C++
Editor
Diff View
More
Share
C++ source #1
void wipe1(int arr[])
{
arr[1] = 0;
}
struct point {
int x, y, z;
};
void wipe2(struct point *ptr)
{
ptr->y = 0;
}
ARM gcc 5.4 (Editor #1, Compiler #1)
ARM gcc 5.4
-Og -ffreestanding -marm
11010
.LX0:
.text
//
Intel
wipe1(int*):
mov r3, #0
str r3, [r0, #4]
bx lr
wipe2(point*):
mov r3, #0
str r3, [r0, #4]
bx lr

Excerpted from blink.s

```
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
```

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

delay:
    mov r2, #0x3F0000
    wait:
        subs r2, #1
        bne wait
// but...  where to go next?
```

```
loop:
   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, r2, #1
     bne wait
   mov pc, r14
```

We've just invented our own link register!

```
    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've just invented our own parameter passing!

Anatomy of C function call

```c
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 <u>caller</u> of** `binky`

`binky` **is <u>callee</u> of** `main`

+ <u>caller</u> of `winky`

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

void binky(int a) {
   winky(10, a);
}

int winky(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 the callee needs scratch space for an intermediate value, which type of register should it choose?

2. What must a callee 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?

The stack to the rescue!

Region in memory to store local variables, scratch space, <u>save register values</u>

- **LIFO: push adds value on top of stack, pop removes lastmost value**
- `r13` **(alias** `sp`**) points to topmost value**
- **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)
{
    binky(3);
}

int binky(int a)
{
    int arr[100];
    return winky(arr, 100);
}
```

Not to scale

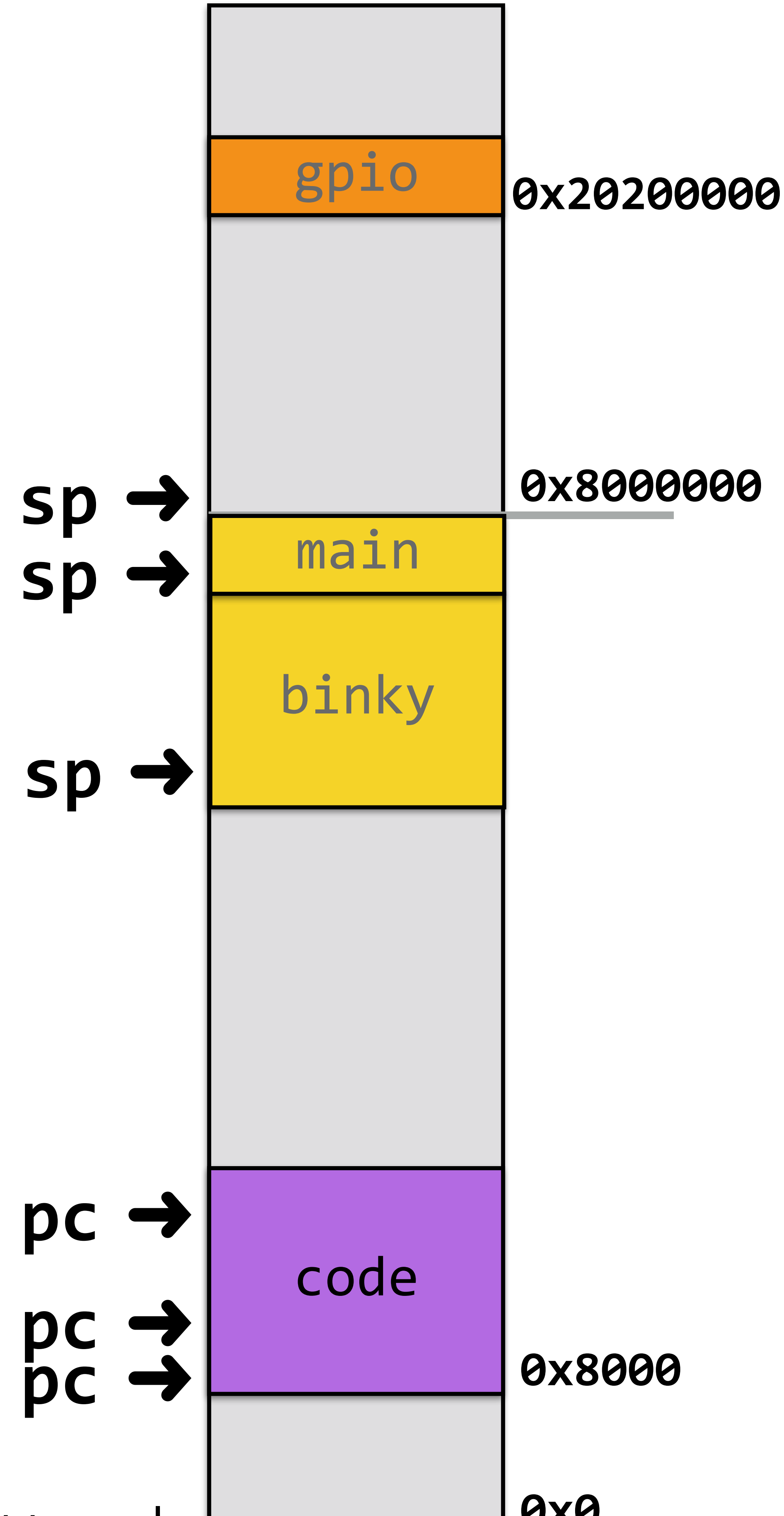
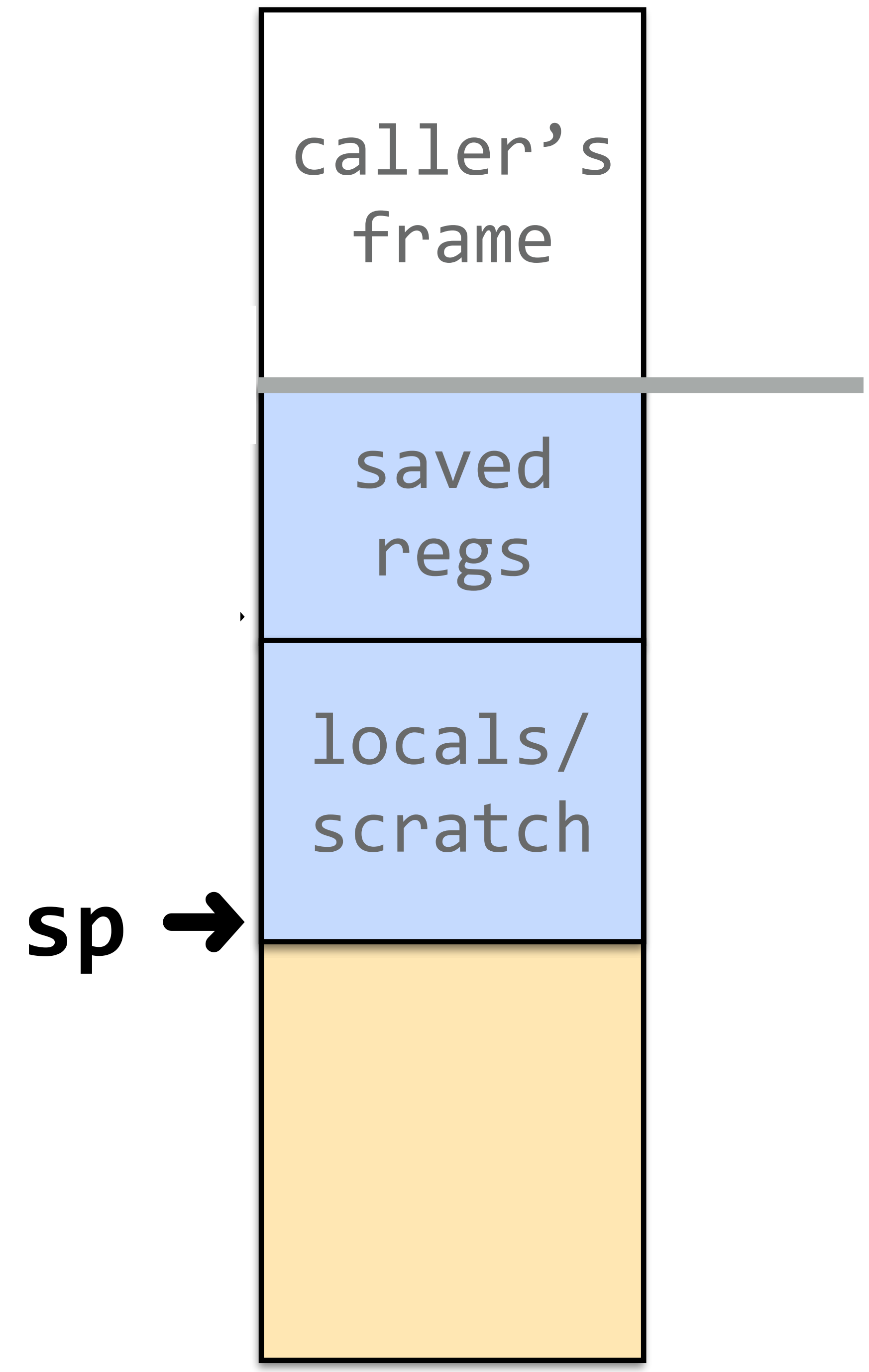


Diagram not to scale

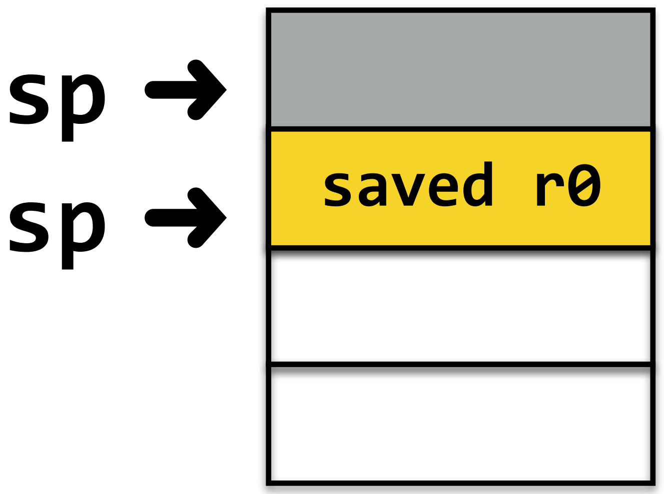
Single stack frame

```
int winky(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

```c
int winky(int a, int b)
{
  int c = binky(a);
  return b + c;
}
```

If winky calls binky...

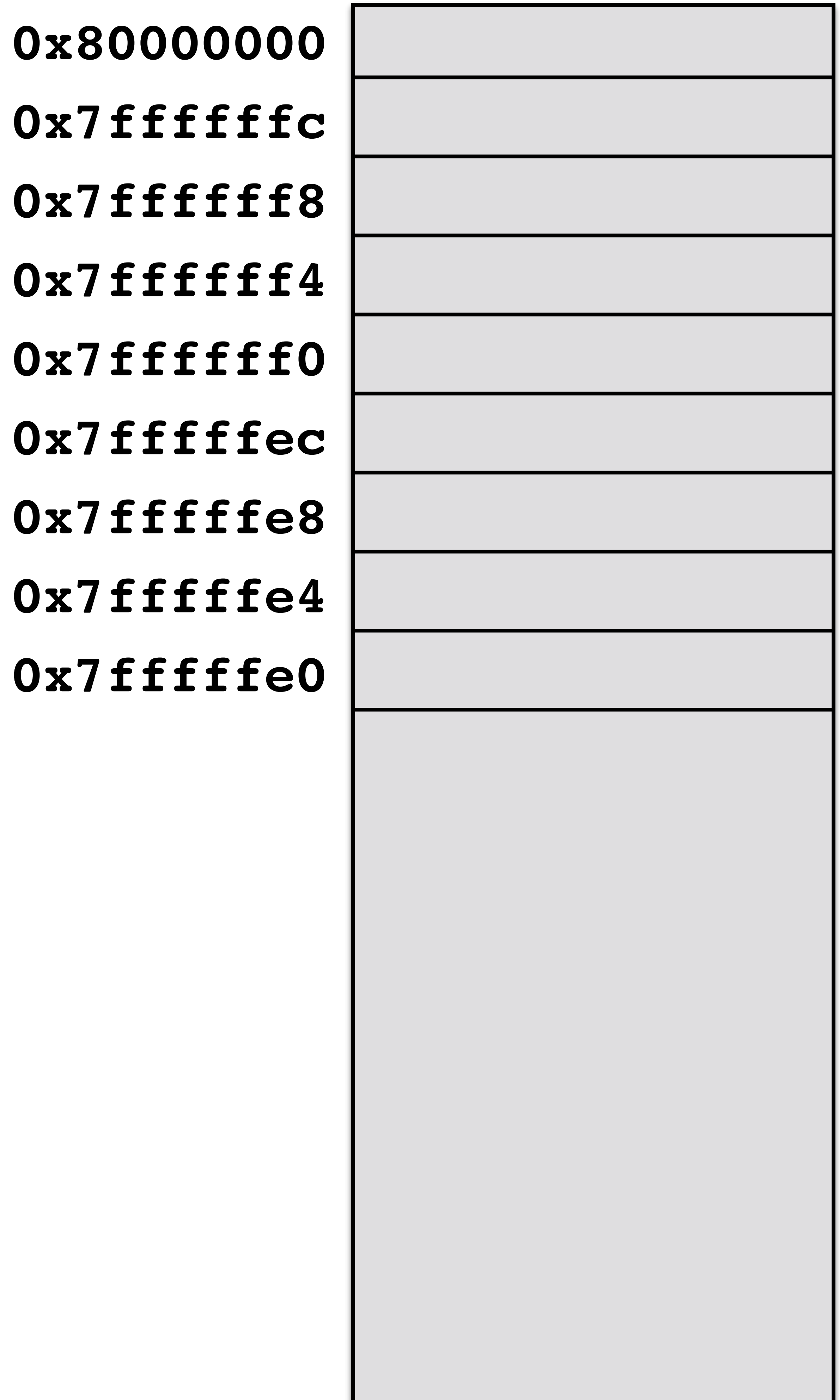
Why do they collide on use of `lr` ?

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

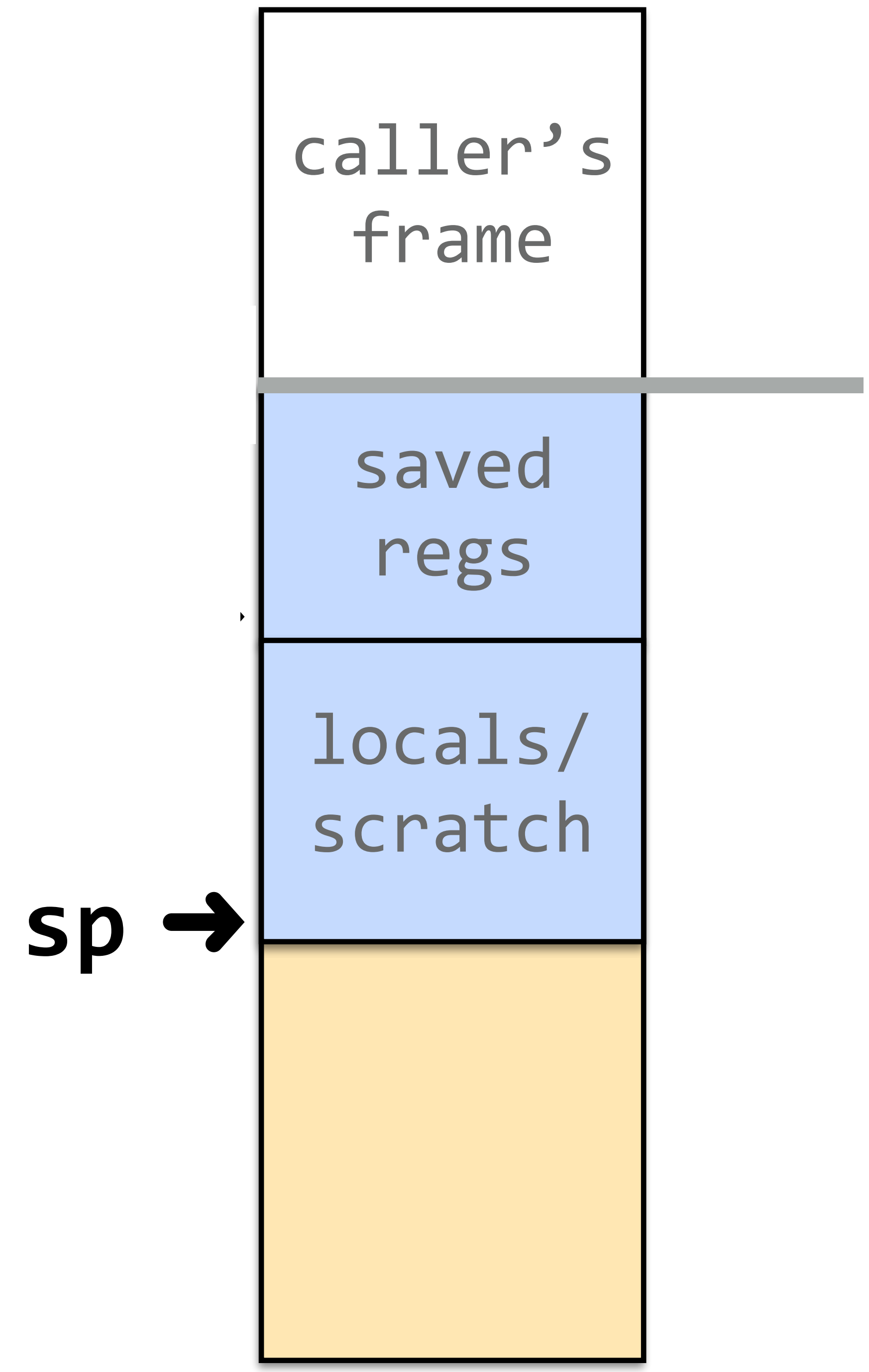
r0
r1
r2
r3
lr
sp
pc

0x80000000	
0x7ffffffc	
0x7ffffff8	
0x7ffffff4	
0x7ffffff0	
0x7fffffec	
0x7fffffe8	
0x7fffffe4	
0x7fffffe0	
⋮	⋮
0x7ffffe70	
0x7ffffe6c	
0x7ffffe68	
0x7ffffe64	
0x7ffffe60	
0x7ffffe5c	
0x7ffffe58	
0x7ffffe54	

sp in constant motion

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

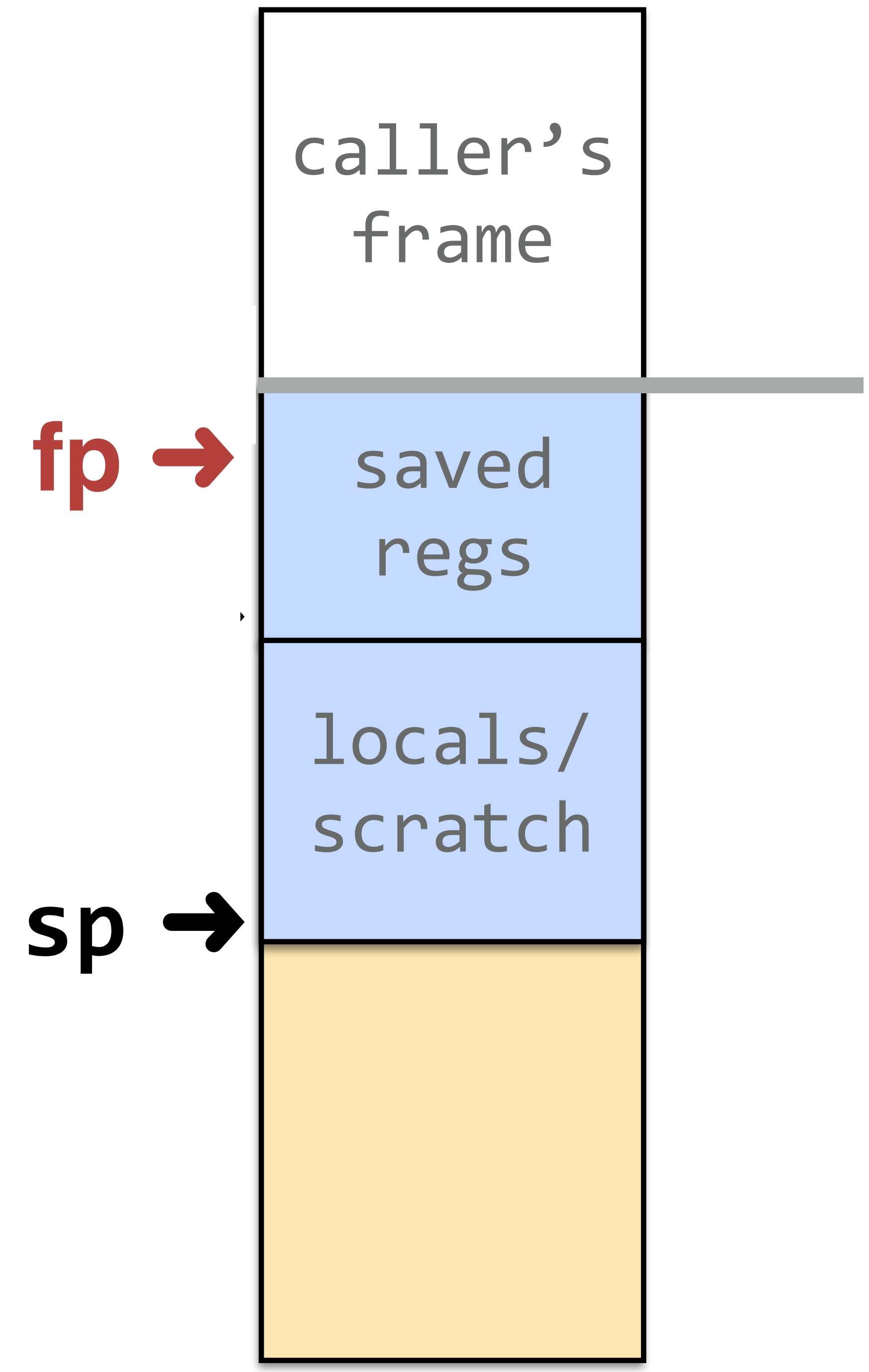
sp is constantly changing!
(push, pop, add sp, sub sp)



Add frame pointer (fp)

Dedicate `fp` register to be used as fixed anchor

Offsets relative to `fp` stay constant!



APCS “full frame”

APCS = <u>A</u>RM <u>P</u>rocedure <u>C</u>all <u>S</u>tandard

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 prolog/epilog to each function that sets up/tears down the standard frame and manages fp

Trace APCS

Prolog

`push fp, r13, lr, pc`

set `fp` to first word of stack frame

Body

`fp` stays anchored

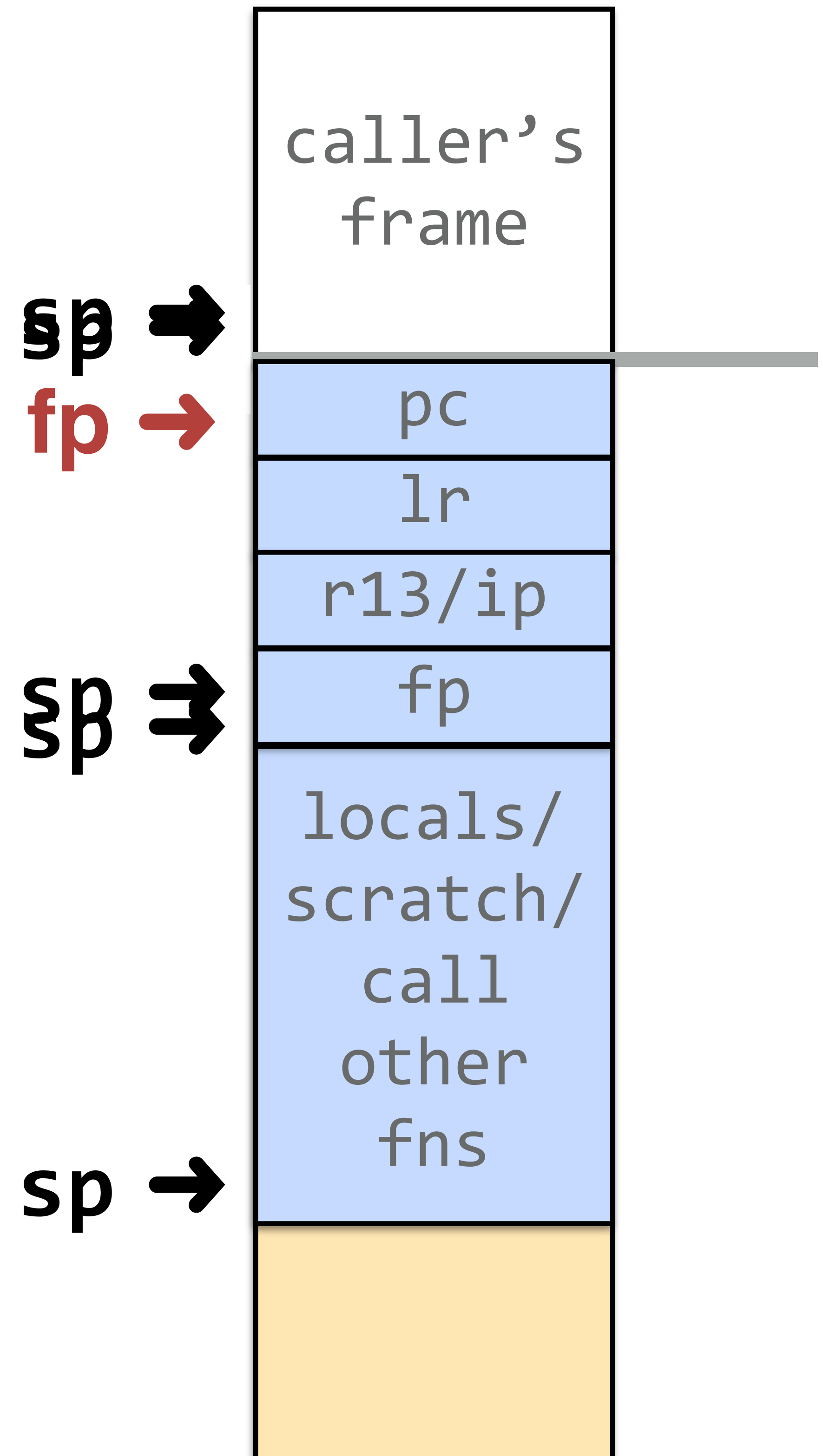
access data on stack `fp`-relative

offsets won't vary even if `sp` changing

Epilog

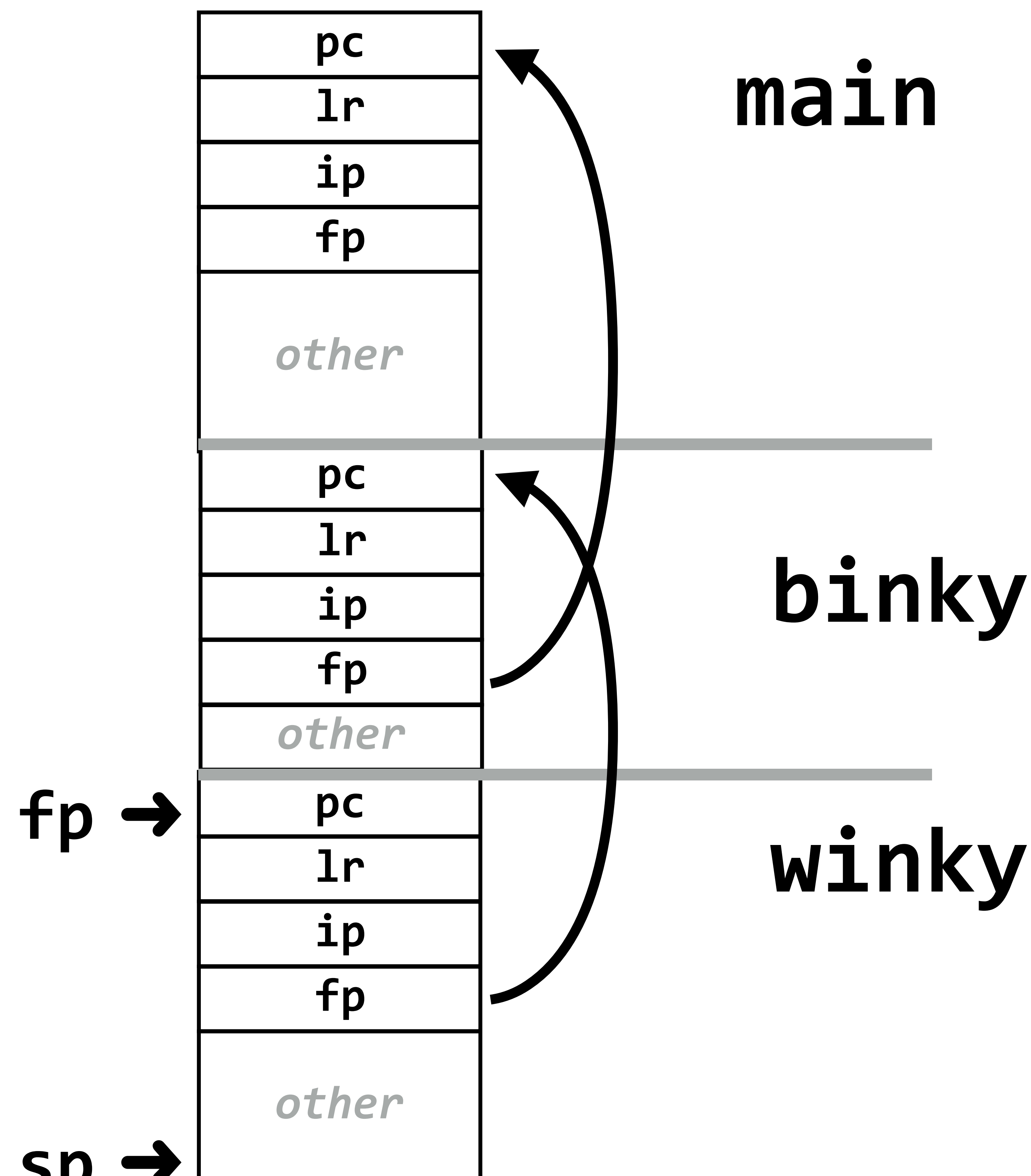
`pop fp, r13, lr`

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