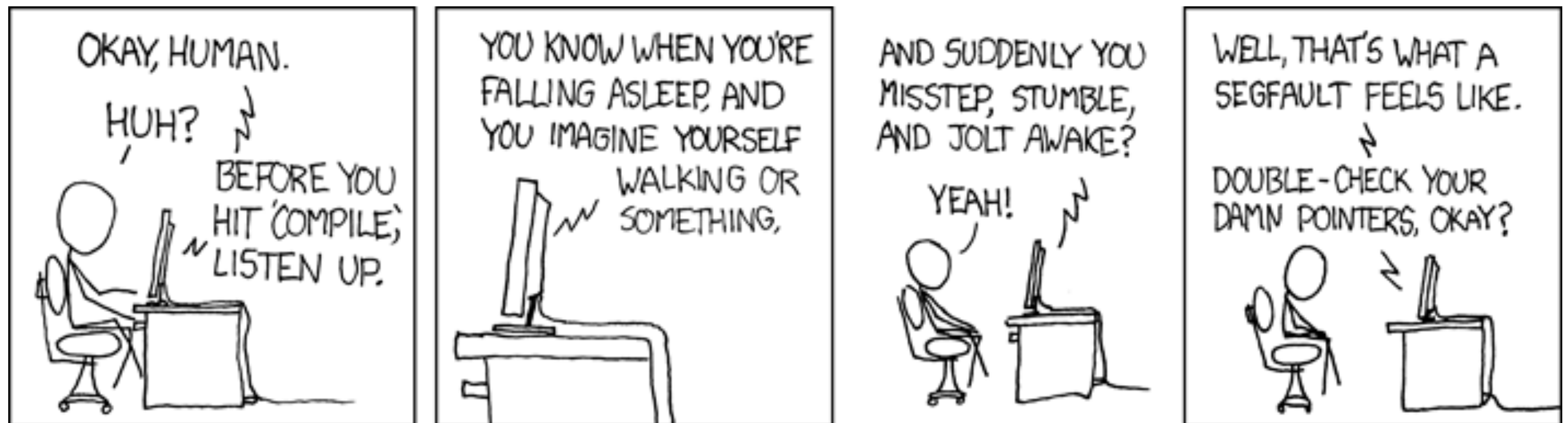


# Goals for today

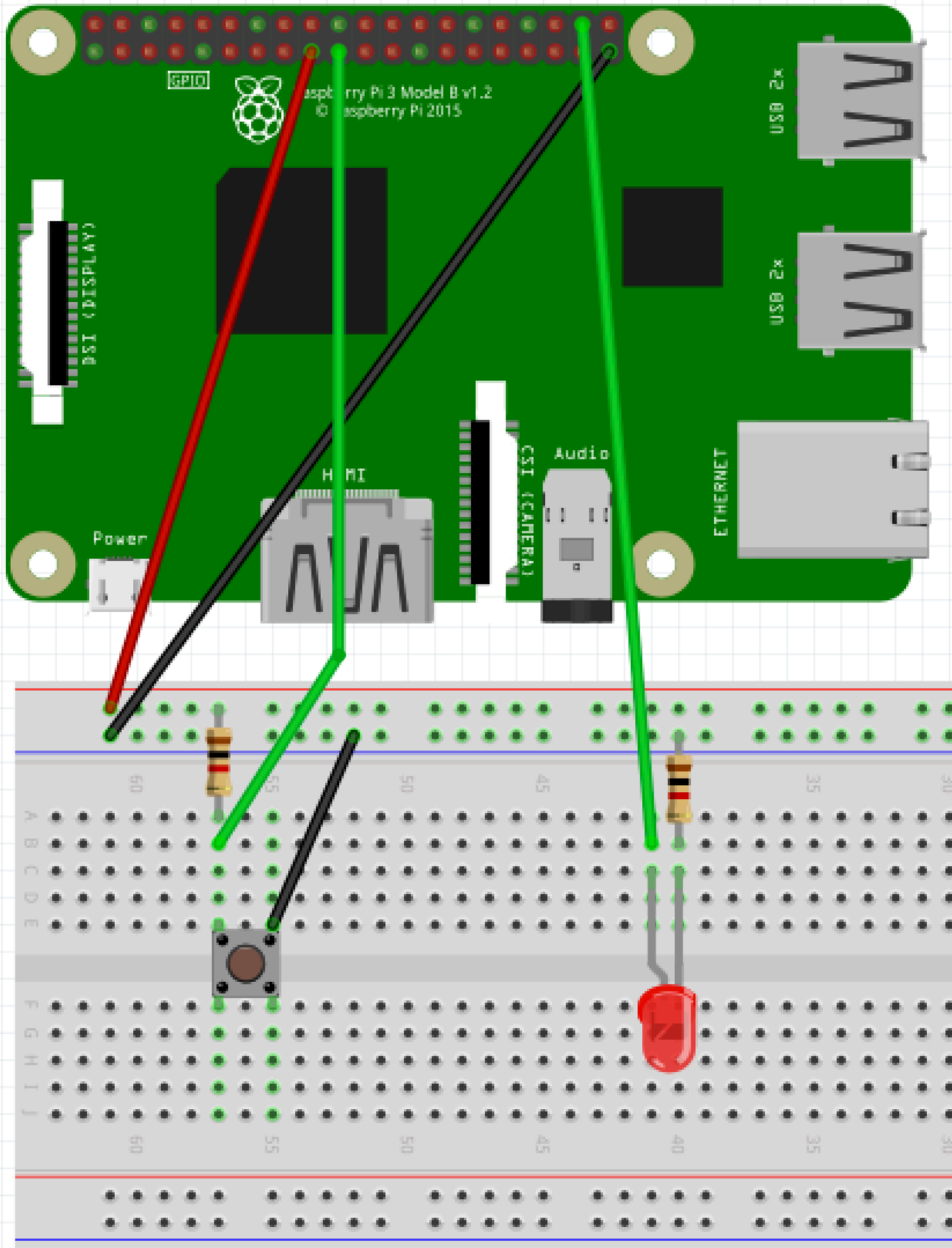
- The little button that wouldn't :(ul>  - the `volatile` keyword
- Pointer operations => ARM addressing modes
- Implementation of C function calls
- Management of runtime stack, register use



**button.c**

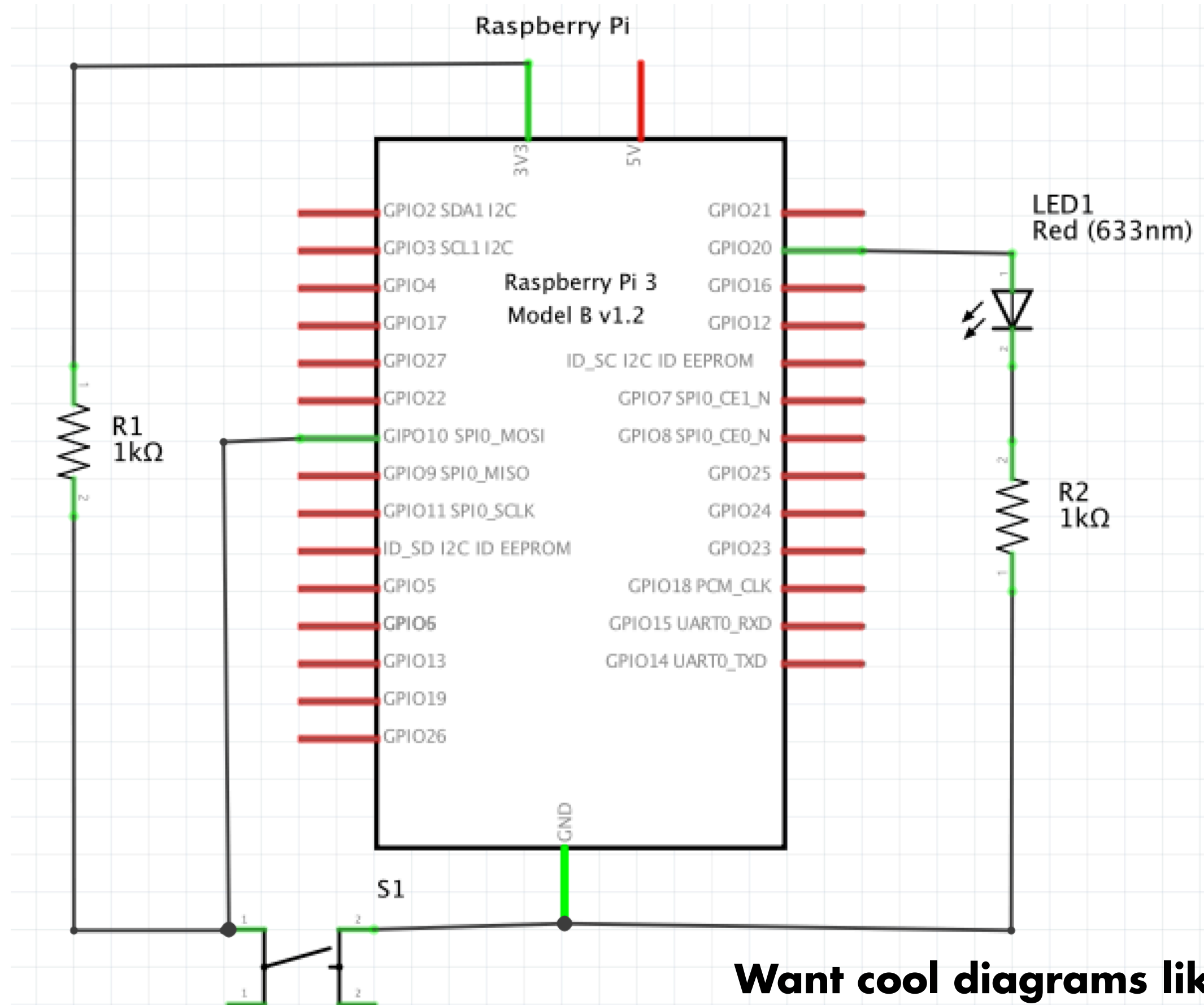
**The little button that wouldn't**

# button.c: The little button that wouldn't



**Want cool diagrams like this?  
Check out [fritzing.org](http://fritzing.org)**

# button.c: The little button that wouldn't



Want cool diagrams like this?  
Check out [fritzing.org](http://fritzing.org)

# button.c: The little button that wouldn't

```
// This program waits until a button is pressed (GPIO 10)
// and turns on GPIO 20.

static unsigned int *FSEL1 = (unsigned int *)0x20200004;
static unsigned int *FSEL2 = (unsigned int *)0x20200008;
static unsigned int *SET0 = (unsigned int *)0x2020001c;
static unsigned int *LEV0 = (unsigned int *)0x20200034;

void main(void)
{
    unsigned int bit_10 = 1 << 10;
    unsigned int bit_20 = 1 << 20;

    *FSEL1 = 0; // bit 10 is input pin
    *FSEL2 = 1; // bit 20 is output pin

    // Wait until GPIO 10 is low (button press)
    while ((*LEV0 & bit_10) != 0) ;

    // Set GPIO 20 high (LED on)
    *SET0 = bit_20;
}
```

# button.c: The little button that wouldn't

```
// This program waits until a button is pressed (GPIO 10)
// and turns on GPIO 20.

static unsigned int *FSEL1 = (unsigned int *)0x20200004;
static unsigned int *FSEL2 = (unsigned int *)0x20200008;
static unsigned int *SET0 = (unsigned int *)0x2020001c;
static unsigned int *LEVO = (unsigned int *)0x20200034;

void main(void)
{
    unsigned int bit_10 = 1 << 10;
    unsigned int bit_20 = 1 << 20;

    *FSEL1 = 0; // bit 10 is input pin
    *FSEL2 = 1; // bit 20 is output pin

    // Wait until GPIO 10 is low (button press)
    while ((*LEVO & bit_10) != 0) ;

    // Set GPIO 20 high (LED on)
    *SET0 = bit_20;
}
```

## Compiling with -O2:

### Disassembly of section .text.startup:

```
00000000 <main>:
   0:    e59f3020  ldr    r3, [pc, #32] ; 28 <main+0x28>
   4:    e5930034  ldr    r0, [r3, #52] ; 0x34
   8:    e3a02001  mov    r2, #1
  c:    e3100b01  tst    r0, #1024 ; 0x400
 10:    e3a01000  mov    r1, #0
 14:    e9830006  stmib  r3, {r1, r2}
 18:    03a02601  moveq  r2, #1048576 ; 0x100000
 1c:    0583201c  streq  r2, [r3, #28]
 20:    012ffffe  bxeq  lr
 24:    eaffffff  b     24 <main+0x24>
 28:    20200000  .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.

static unsigned int *FSEL1 = (unsigned int *)0x20200004;
static unsigned int *FSEL2 = (unsigned int *)0x20200008;
static unsigned int *SET0 = (unsigned int *)0x2020001c;
static unsigned int *LEVO = (unsigned int *)0x20200034;

void main(void)
{
    unsigned int bit_10 = 1 << 10;
    unsigned int bit_20 = 1 << 20;

    *FSEL1 = 0; // bit 10 is input pin
    *FSEL2 = 1; // bit 20 is output pin

    // Wait until GPIO 10 is low (button press)
    while ((*LEVO & bit_10) != 0) ;

    // Set GPIO 20 high (LED on)
    *SET0 = bit_20;
}
```

## Compiling with -O2:

Disassembly of section .text.startup:

```
00000000 <main>:
   0:   e59f3020  ldr    r3, [pc, #32] ; 28 <main+0x28>
   4:   e5930034  ldr    r0, [r3, #52] ; 0x34
   8:   e3a02001  mov    r2, #1
  c:   e3100b01  tst    r0, #1024 ; 0x400
 10:   e3a01000  mov    r1, #0
 14:   e9830006  stmib  r3, {r1, r2}
 18:   03a02601  moveq  r2, #1048576 ; 0x100000
 1c:   0583201c  streq  r2, [r3, #28]
 20:   012fff1e  bxeq  lr
 24:   eaffffff  b     24 <main+0x24>
 28:   20200000  .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.

static unsigned int *FSEL1 = (unsigned int *)0x20200004;
static unsigned int *FSEL2 = (unsigned int *)0x20200008;
static unsigned int *SET0 = (unsigned int *)0x2020001c;
static unsigned int *LEVO = (unsigned int *)0x20200034;

void main(void)
{
    unsigned int bit_10 = 1 << 10;
    unsigned int bit_20 = 1 << 20;

    *FSEL1 = 0; // bit 10 is input pin
    *FSEL2 = 1; // bit 20 is output pin

    // Wait until GPIO 10 is low (button press)
    while ((*LEVO & bit_10) != 0) ;

    // Set GPIO 20 high (LED on)
    *SET0 = bit_20;
}
```

**What happened to the loop? Why are things out of order?**

## Compiling with -O2:

Disassembly of section .text.startup:

```
00000000 <main>:
 0: e59f3020 ldr r3, [pc, #32] ; 28 <main+0x28>
 4: e5930034 ldr r0, [r3, #52] ; 0x34
 8: e3a02001 mov r2, #1
 c: e3100b01 tst r0, #1024 ; 0x400
10: e3a01000 mov r1, #0
14: e9830006 stmib r3, {r1, r2}
18: 03a02601 moveq r2, #1048576 ; 0x100000
1c: 0583201c streq r2, [r3, #28] ?
20: 012fff1e bxeq lr ?
24: eaffffff b 24 <main+0x24>
28: 20200000 .word 0x20200000
```



# 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: 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.**

# 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

```
LD_FLAGS = -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)**

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 -ffreestanding -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, 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 is ARM embedded ABI**  
**("none" refers to no hosting OS)**



# 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 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!

Reserve section of memory to store data for executing functions

Stack frame allocated per function invocation

Can store local variables, scratch values, saved registers

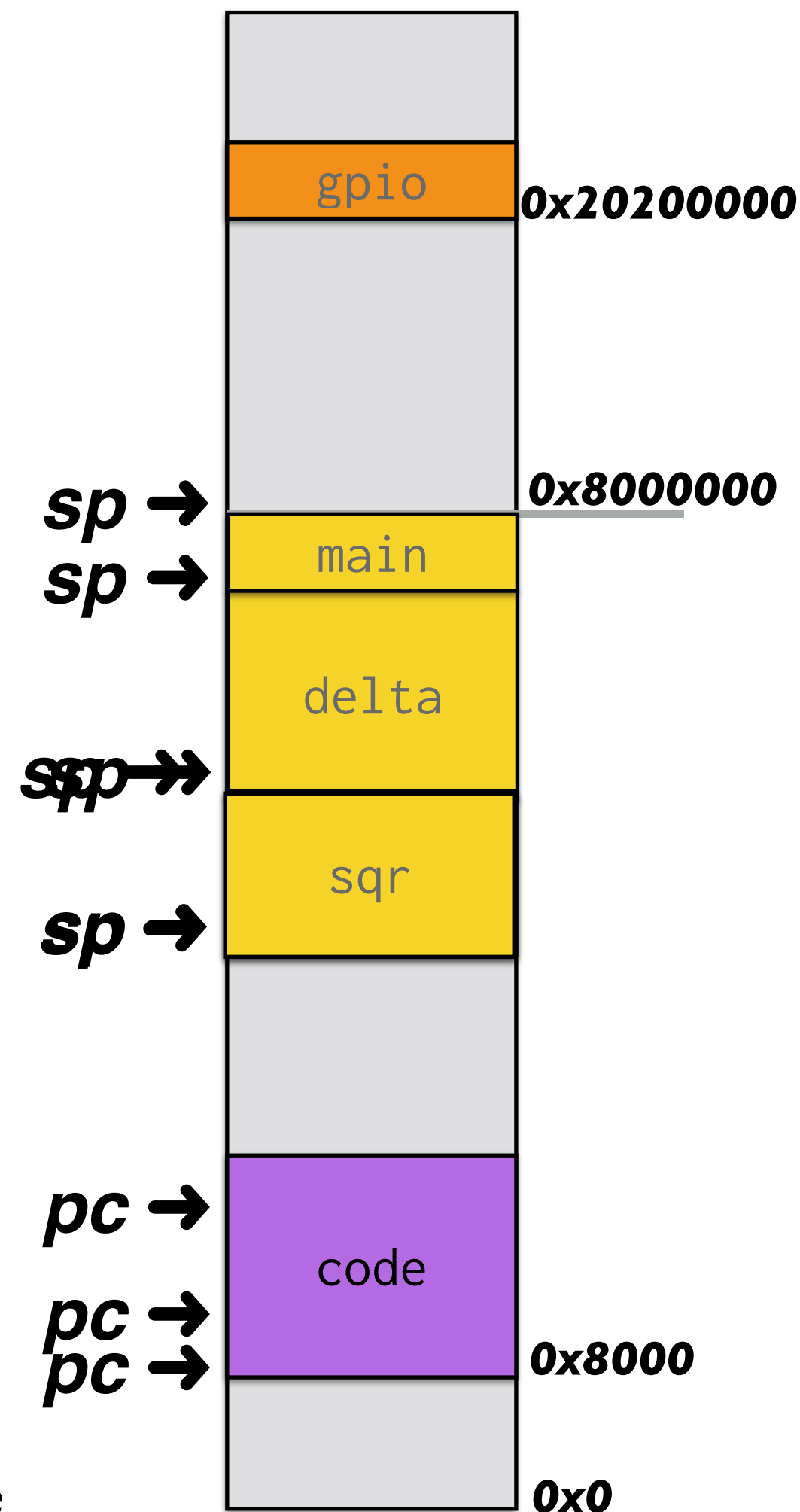
- LIFO: **push** adds value on top of stack, **pop** removes lastmost value
- **r13** (alias **sp**) points to lastmost value pushed
- stack grows down
  - newer values at lower addresses
  - push subtracts from **sp**
  - pop adds to **sp**
- **push/pop** aliases for load/store multiple with writeback

```
// start.s
mov sp, #0x80000000
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

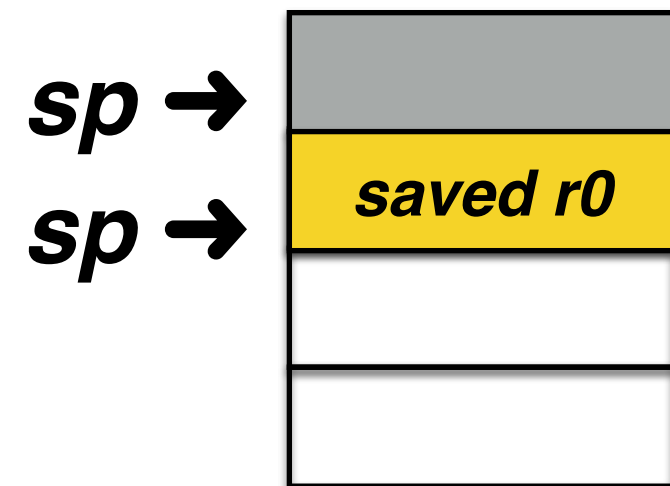
```
// push to saved reg val on stack  
// *--sp = r0  
// decrement sp before store  
// equivalent: str r0, [sp, #-4]!
```

**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 8-byte aligned, always push/pop 2, 4, 6,... (e.g. even) number of registers

# 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

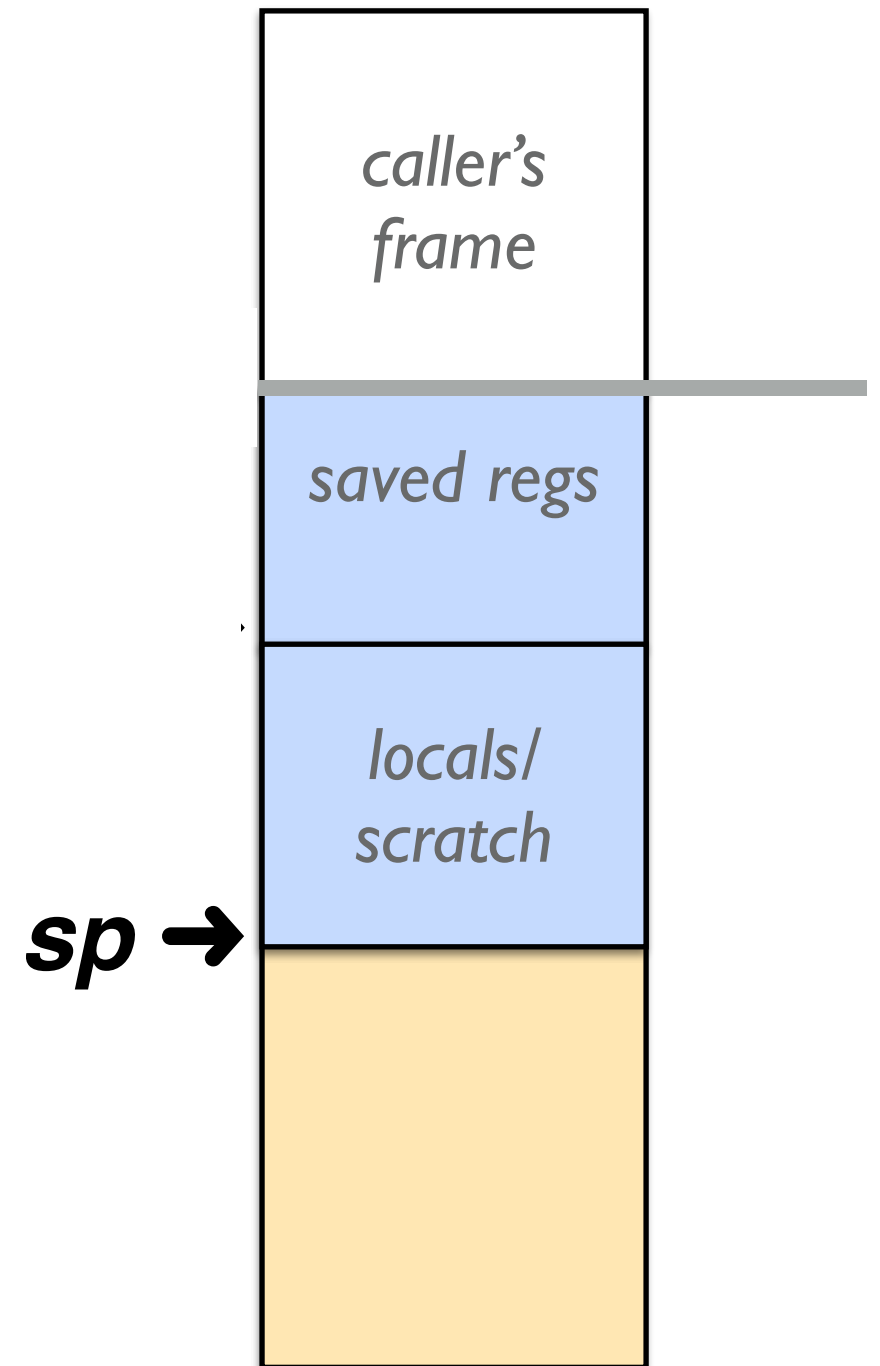
```
$ arm-none-eabi-gdb program.elf  
(gdb) target sim  
(gdb) load
```

**Read our guide to gdb simulation**  
**<http://cs107e.github.io/guides/gdb/>**

# sp in constant motion

Could access values on stack using **sp**-relative addressing, but .....

**sp** is constantly changing!

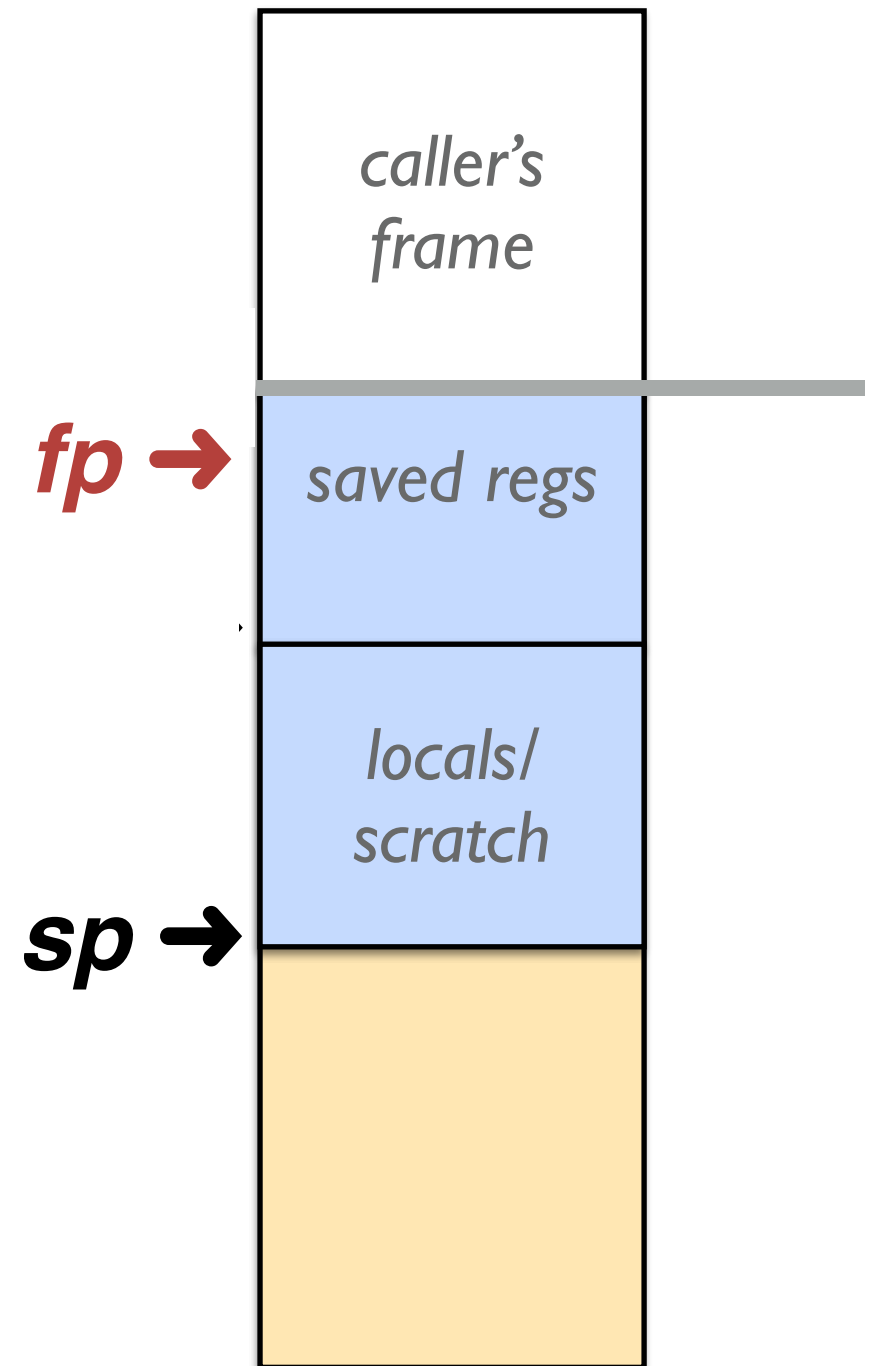


# Add frame pointer

Dedicate **fp** register to be used as fixed anchor

Assign on entry to new function to point to top of stack frame

**fp** doesn't change, can access data at fixed offset relative to **fp**



# APCS “full frame”

APCS = ARM Procedure Call Standard

Conventions for frame pointer and frame layout

Enable reliable stack introspection

CFLAGS to enable: **-mapcs-frame**

**r11** used as **fp**

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

# Trace APCS full frame

## Prolog

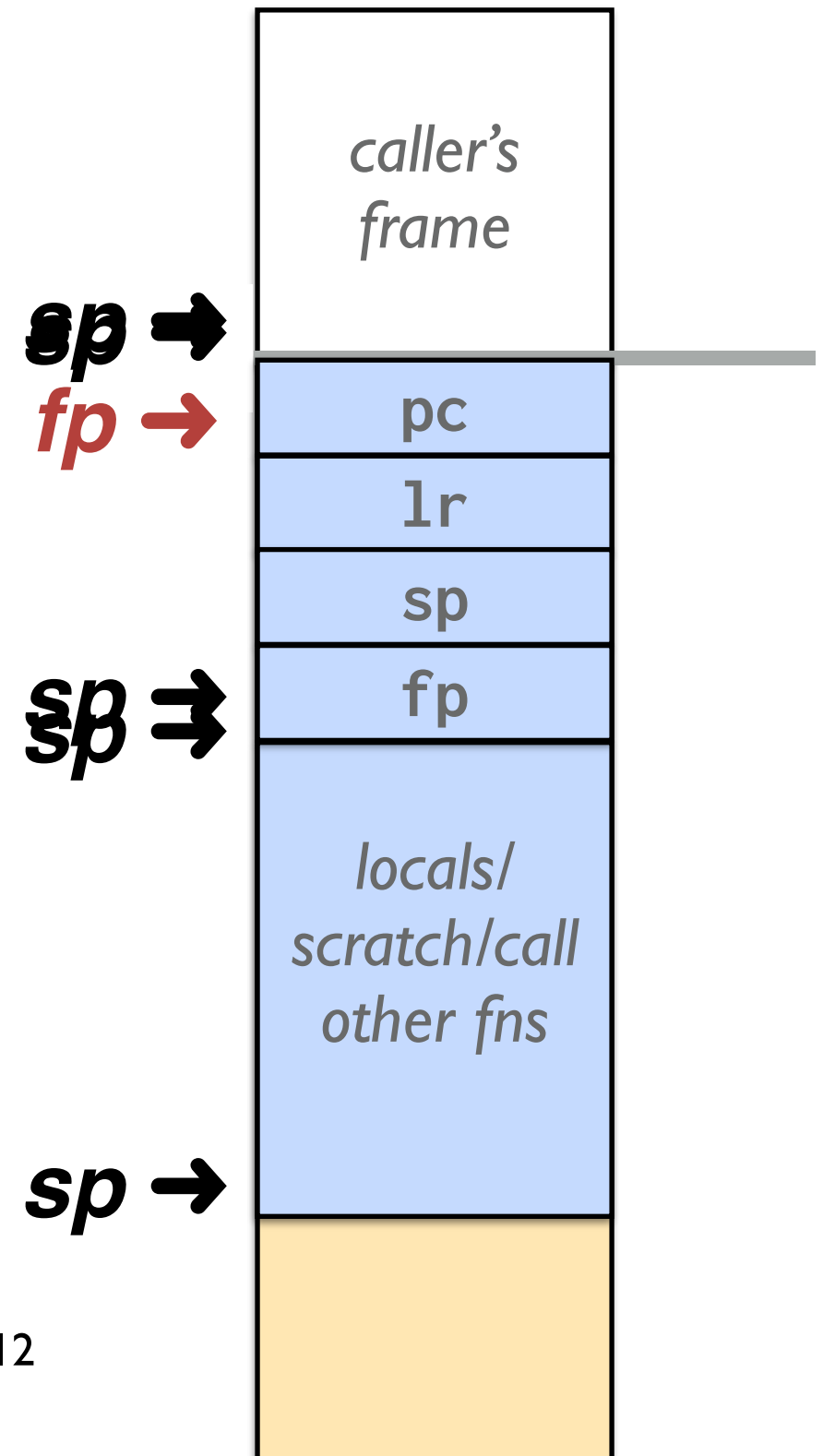
push fp, sp\*, lr, pc  
set **fp** to first word of stack frame

## Body

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

## Epilog

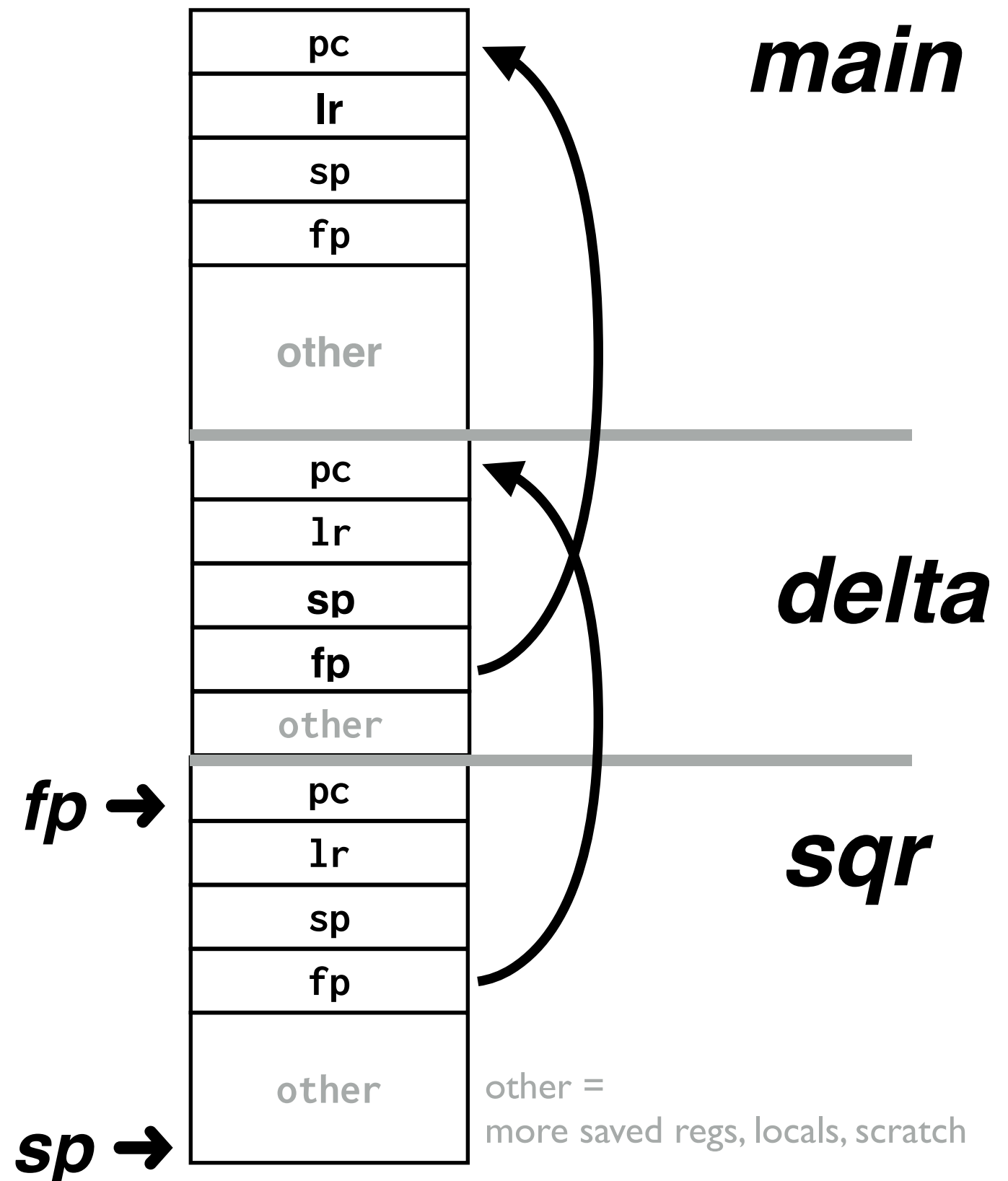
pop fp, sp\*, lr, pc\*



\* 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

# Frame pointers form linked chain

Can start at currently executing call (**sqr**) and back up to caller (**delta**), from there to its caller (**main**), who ends the chain



**// start.s**

**// add init fp = NULL**

**// to terminate end of chain**

**mov sp, #0x80000000**

**mov fp, #0**

**bl main**

# APCS Pros/Cons

- + Anchored fp, offsets are constant
- + Standard frame layout enables runtime introspection
- + Backtrace for debugging
- + Unwind stack on exception
- High overhead cost, every function call affected
- Extra ~5 instructions to setup/tear down frame each call
- 4 registers push/pop => extra 16 bytes per frame
- fp monopolizes use of one of our precious registers