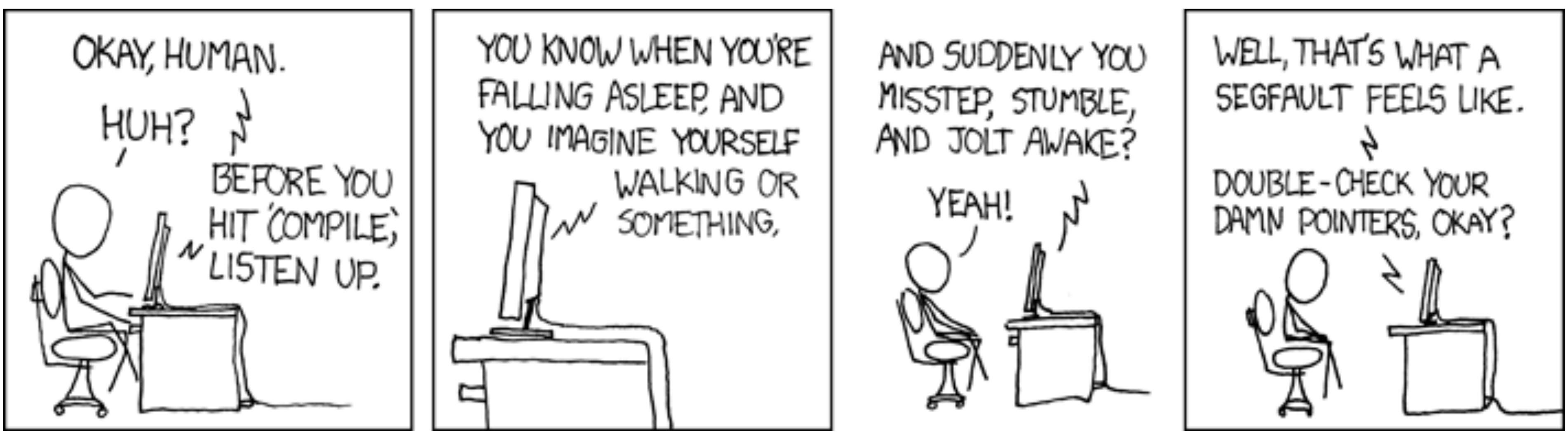
Goals for today

- How pointers ≈ arrays ≈ C-strings
- Implementation of C function calls
- Management of runtime stack, register use



The utility of pointers!

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

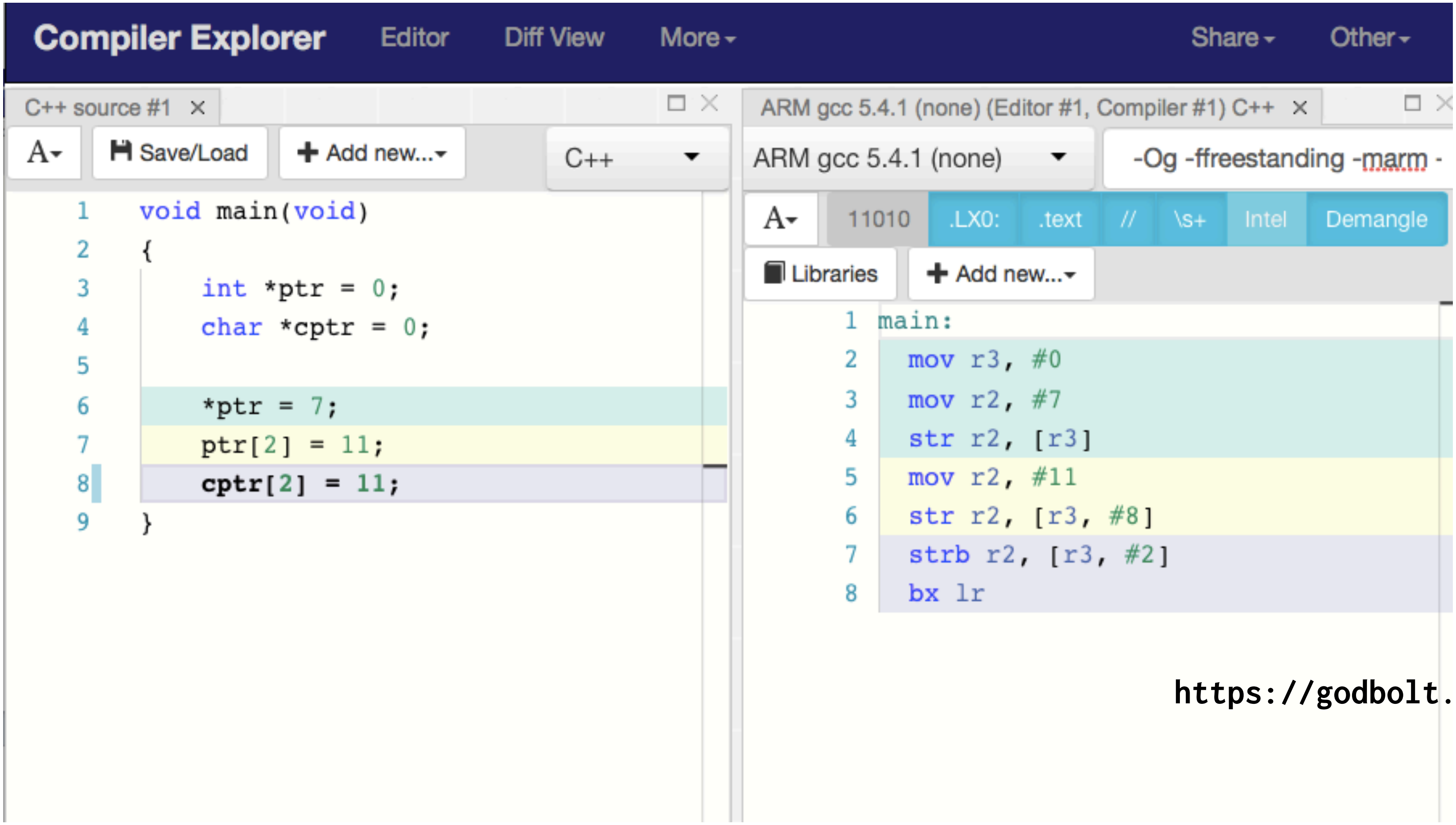
Access data by memory address is essential/useful!

`ldr`, `str` assembly instructions

C pointers give similar functionality, but with added benefits

- Readability, some safety
- Pointee and level of indirection explicit in the type
- Pointer arithmetic to access contiguous data

ARM addressing modes, C data structures go hand in hand…



https://godbolt.org

How does type of "pointee" influence the generated assembly?

Arrays ≈ pointers

Array is just a sequence of elements contiguous in memory

Sequence of memory locations can be treated an array and vice versa

No array "abstraction", no length tracking, no bounds checking

```
int n, arr[4], *p;

p = arr;
p = &arr[0];   // same as prev line!

*p = 3;
p[0] = 3;      // same as prev line!

n = *(arr + 1);
n = arr[1];    // same as prev line!
```

char array ≈ char * ≈ C-strings

No string "abstraction"

Just sequence of chars in memory, terminated by null char

```
char *s = "Leland";
char *t;

t = s;
s[0] = 'R';
t = s + 3;
t[1] = 'x';
```

\0
64
63
61
6c
65
4c

Excerpted from blink.s

```
loop:
   ldr r0, SET0                // turn on
   str r1, [r0]

   mov r2, #0x3F0000          // delay loop
   wait1:
      subs r2, #1
      bne wait1

   ldr r0, CLR0                // turn off
   str r1, [r0]

   mov r2, #0x3F0000          // delay loop
   wait2:
      subs r2, #1
      bne wait2
  b loop
```

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

```
delay:
  mov r2, #0x3F0000
  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

delay:
   mov r2, #0x3F0000
   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, #0x3F0000
    mov r14, pc
    b delay
    ldr r0, CLR0
    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

```
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-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 being called

`main` is <u>caller</u> of `delta`

`delta` is <u>callee</u> of `main`

`delta` is <u>caller</u> of `abs`

```
void main(void) {
   delta(10, 7);
}

int delta(int x, int y) {
   return abs(x-y);
}

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

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

Points to ponder…

1. If callee needs scratch space for an intermediate result, 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? Wouldn’t it be simpler if all were same?

The stack to the rescue!

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 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
```

```
void main(void)
{
    delta(10,7);
}
```

```
int delta(int x, y)
{
  return abs(x - y);
}
```

Not to scale

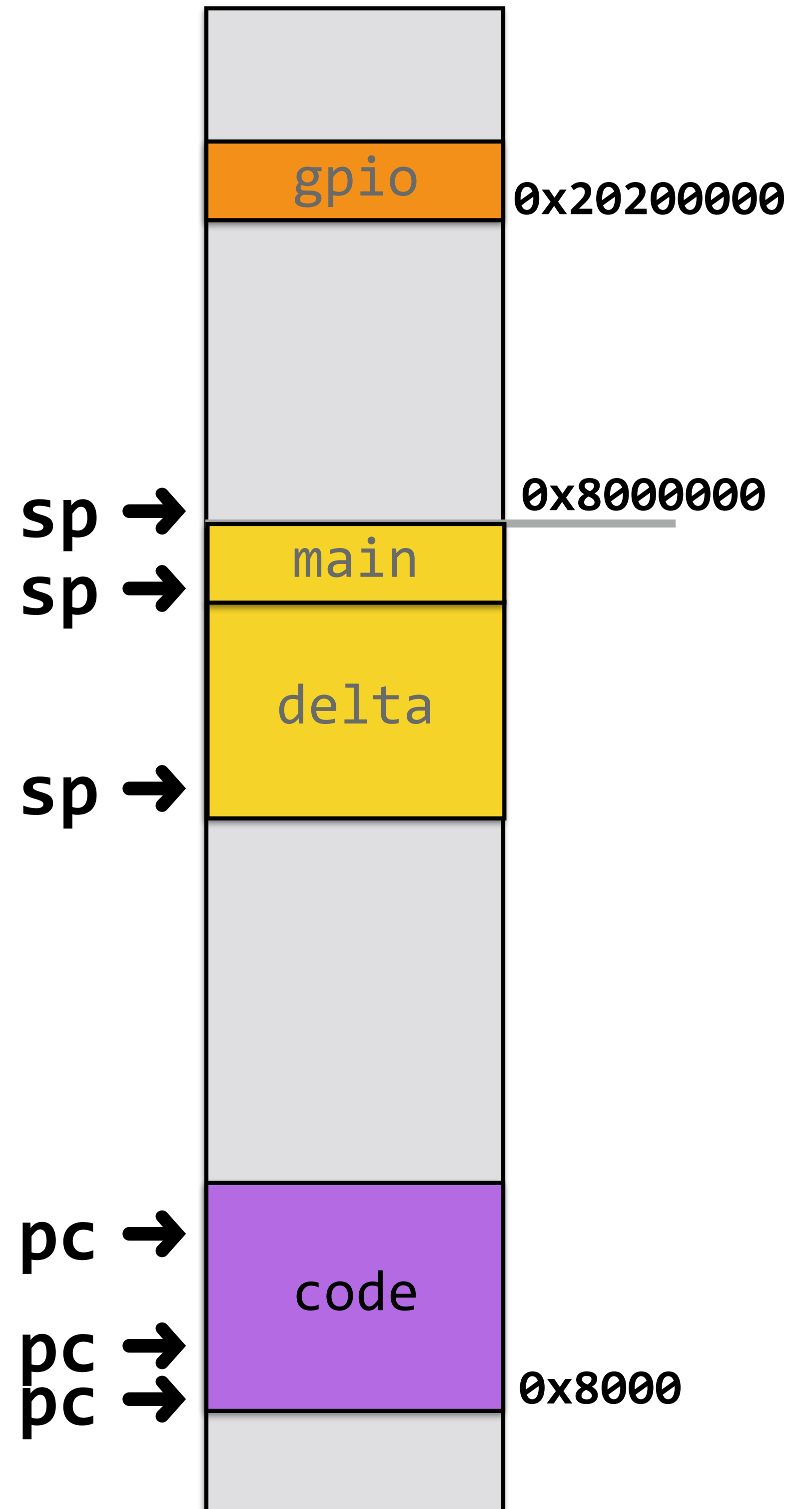
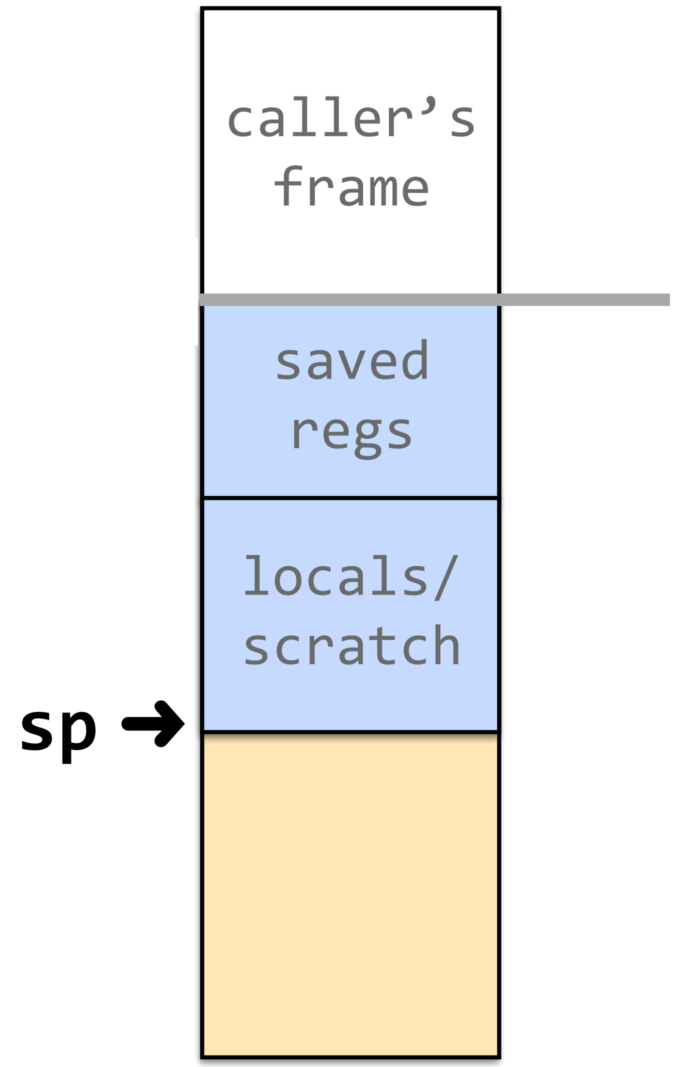


Diagram not to scale

Single stack frame

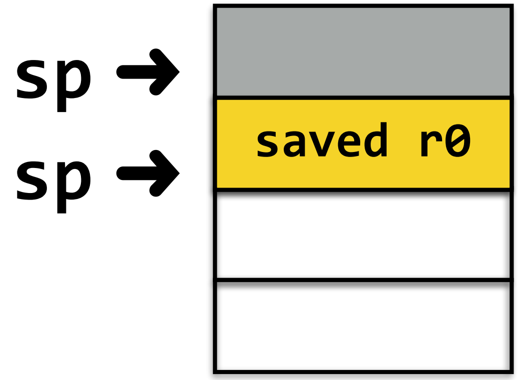
```
int binky(int a, int b)
{
  int c = 2*a;
  ...
  return c;
}
```



Stack operations

```
// PUSH (store reg to stack)
// *--sp = r0
// decrement sp before store
push {r0}
```

```
// POP (restore reg from stack)
// r0 = *sp++
// increment sp after load
pop {r0}
```



"Full Descending" stack

```
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!

Code example: simple.c

Guide to gdb simulation mode

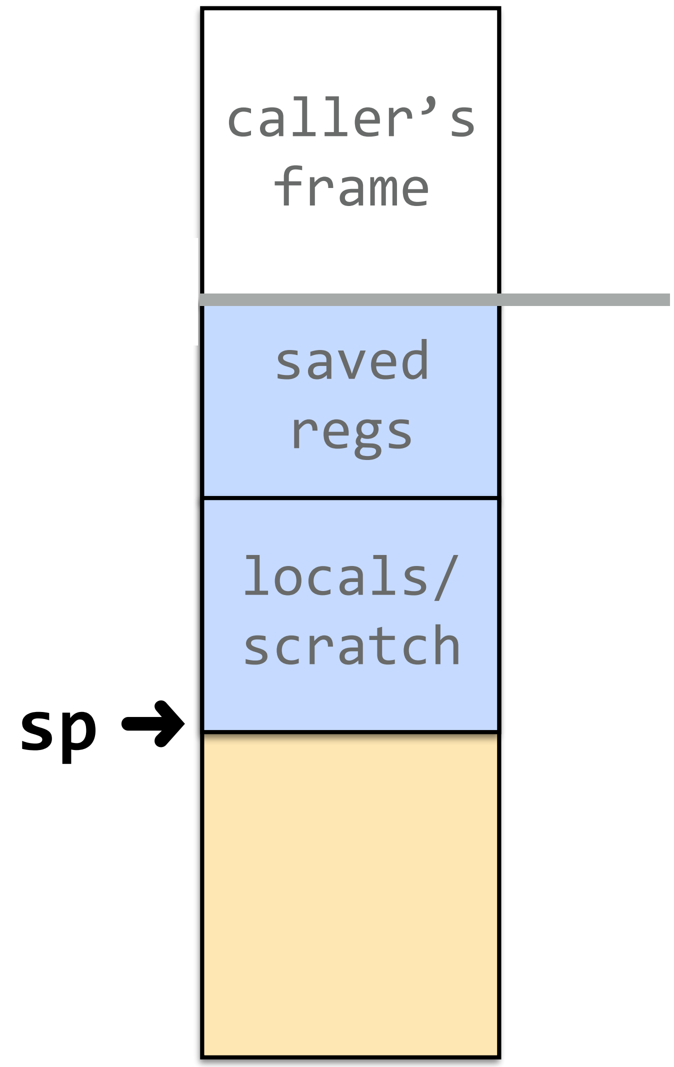
http://cs107e.github.io/guides/gdb/

```
int sum(int n, int m)
{
    return n + m;
}

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

int min(int x, int y)
{
    return x < y ? x : y;
}

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

void main(void)
{
    int p = 33, q = 107;
    assert(sum(p, q) == (min(p, q) + delta(p, q)));
}
```

sp in constant motion

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

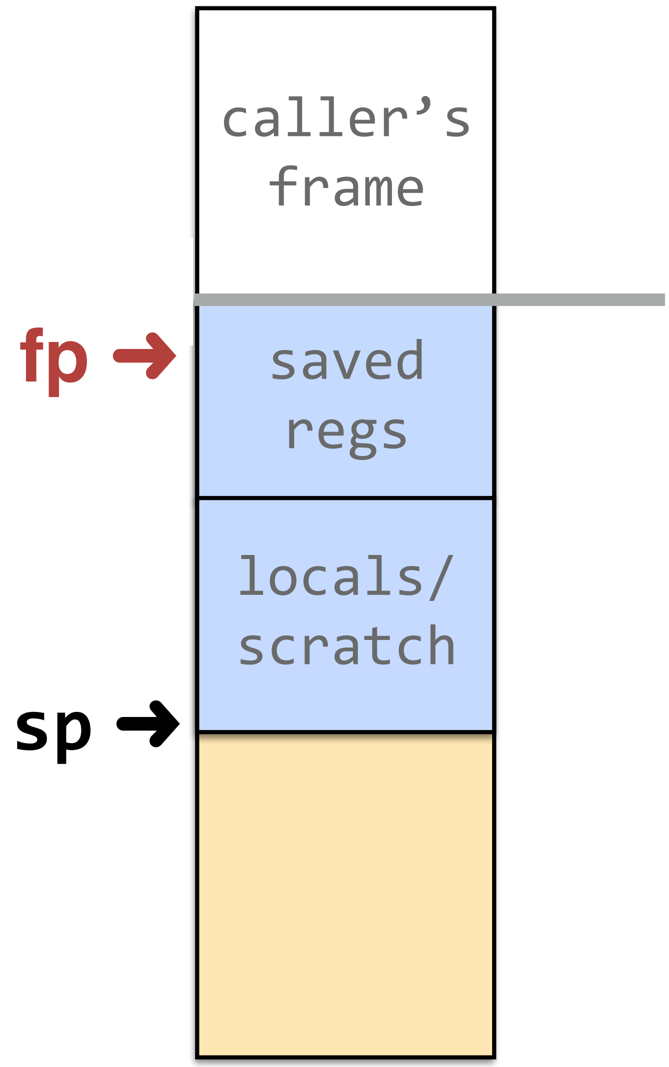
sp is constantly changing!
(push, pop, add sp, sub sp)



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

Trace APCS full frame

Prolog

`push fp, r13, lr, pc`

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

Body

`fp` stays anchored

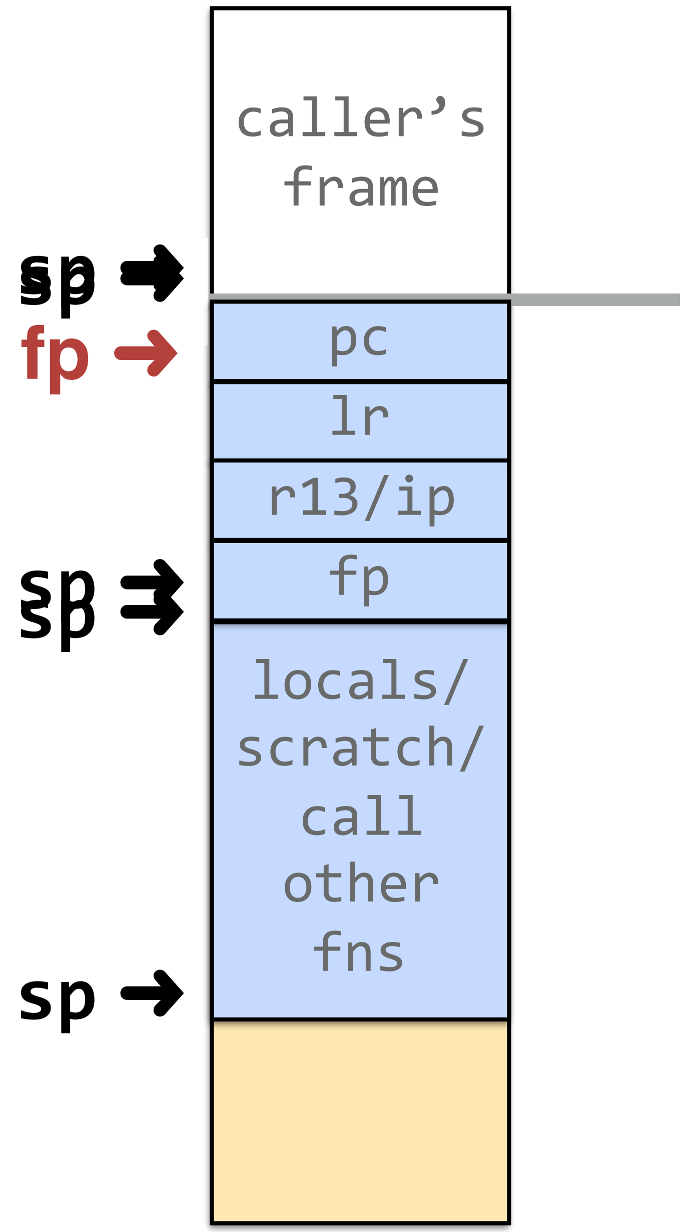
access data on stack **`fp`**-relative

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

Epilog

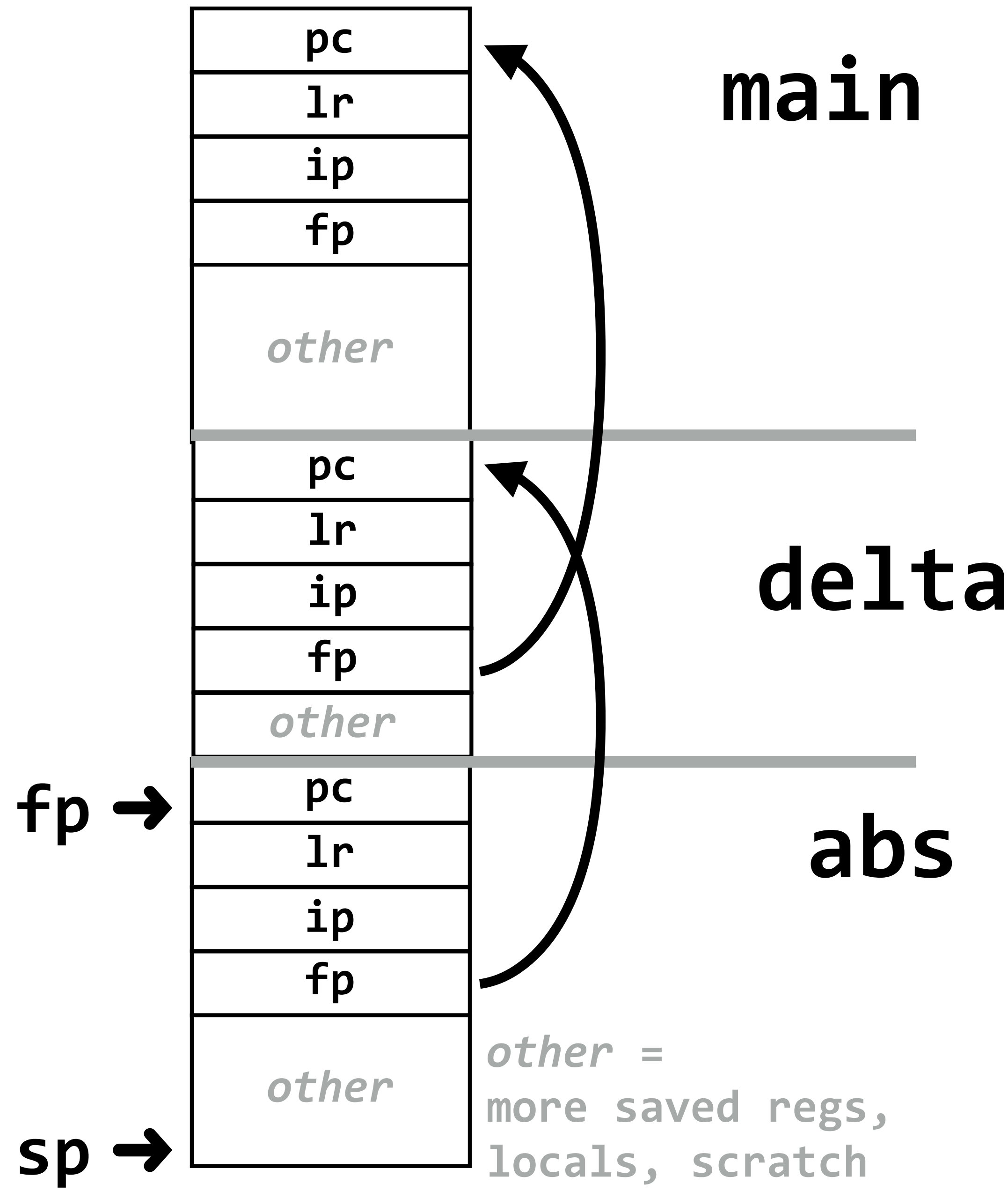
`pop fp, r13, lr`

can't pop **`pc`** (why not?), manually adjust stack



Frame pointers form linked chain

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



```
// 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
+ Standard frame layout enables runtime 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
  - fp consumes one of our precious registers