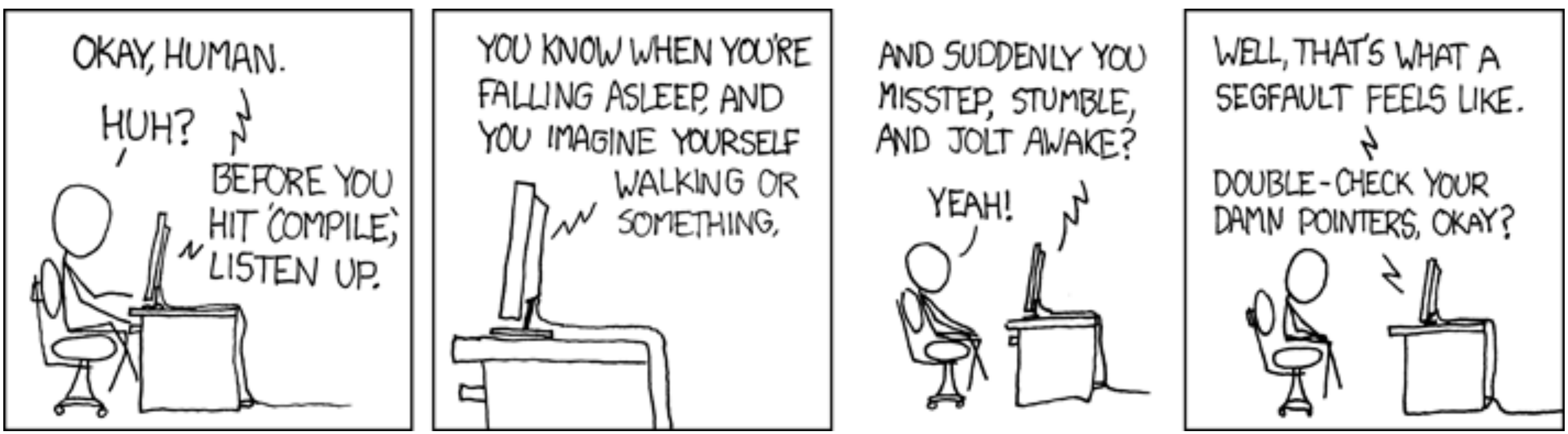
Goals for today

- Pointer operations => ARM addressing modes
- Implementation of C function calls
- Management of runtime stack, register use

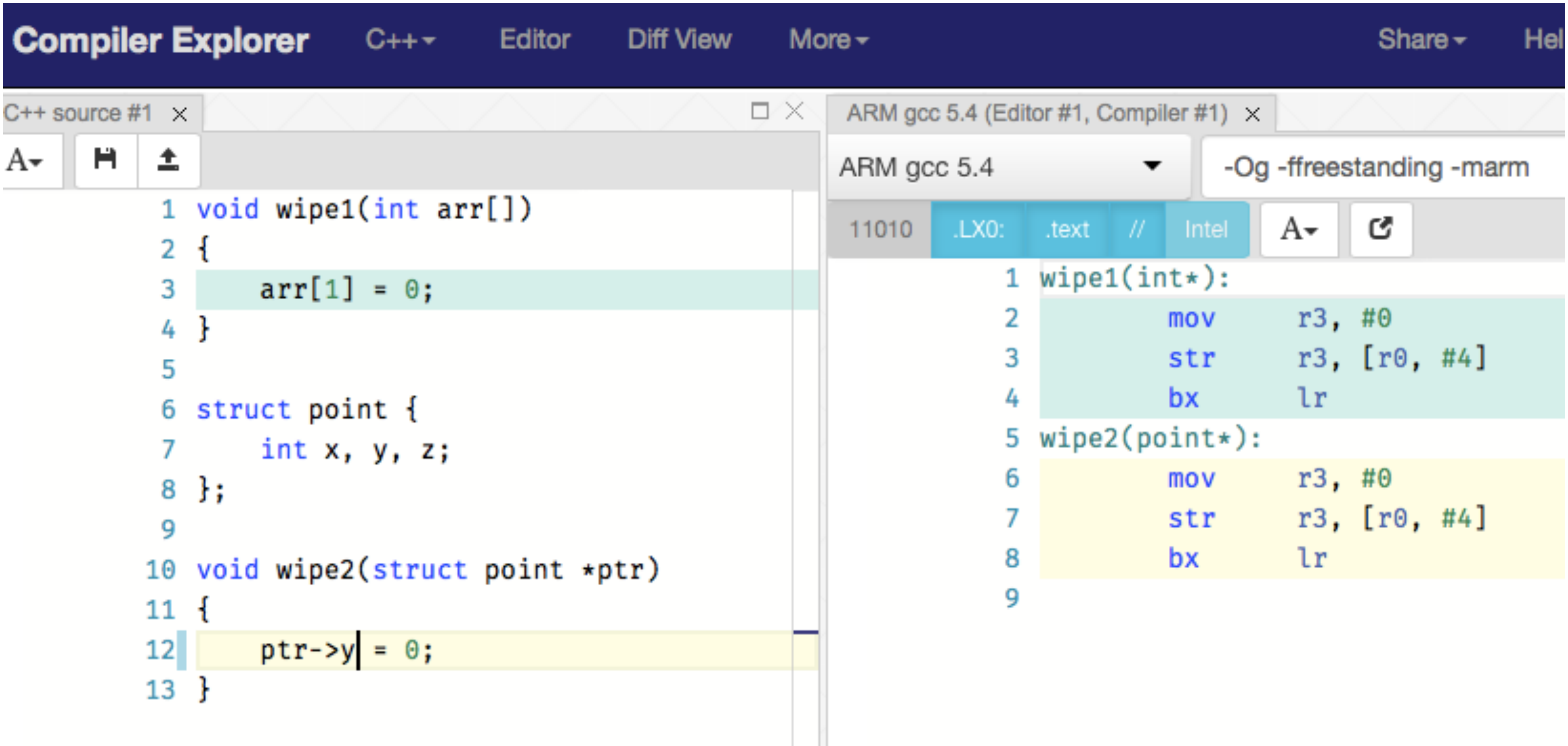


Pointers: more gain than pain!

"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 <u>very useful</u>!

- **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
Hel
C++ source #1
ARM gcc 5.4 (Editor #1, Compiler #1)
ARM gcc 5.4
-Og -ffreestanding -marm
11010
.LX0:
.text
//
Intel
void wipe1(int arr[])
{
    arr[1] = 0;
}

struct point {
    int x, y, z;
};

void wipe2(struct point *ptr)
{
    ptr->y = 0;
}
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?
```

```
    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
```

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

```
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

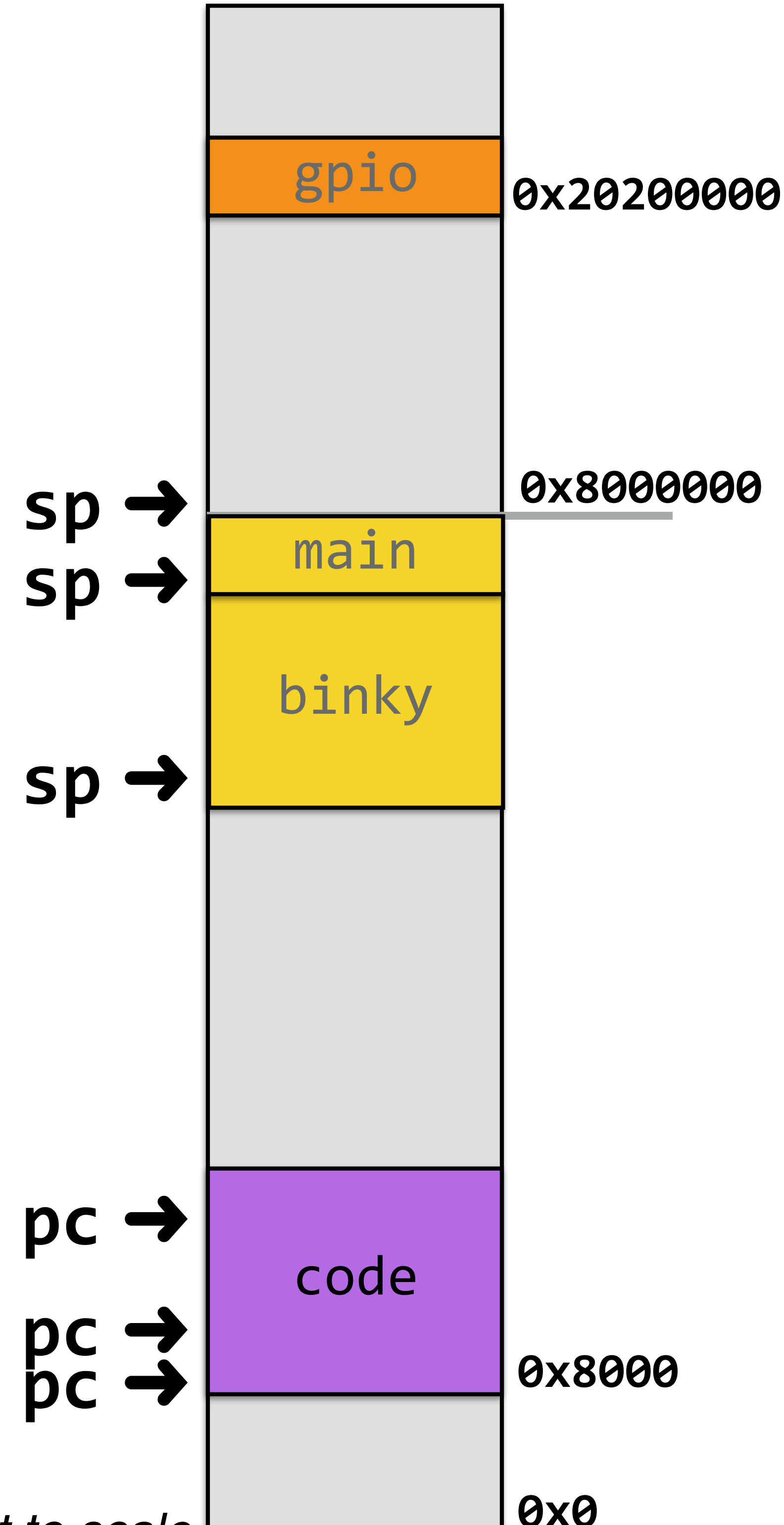
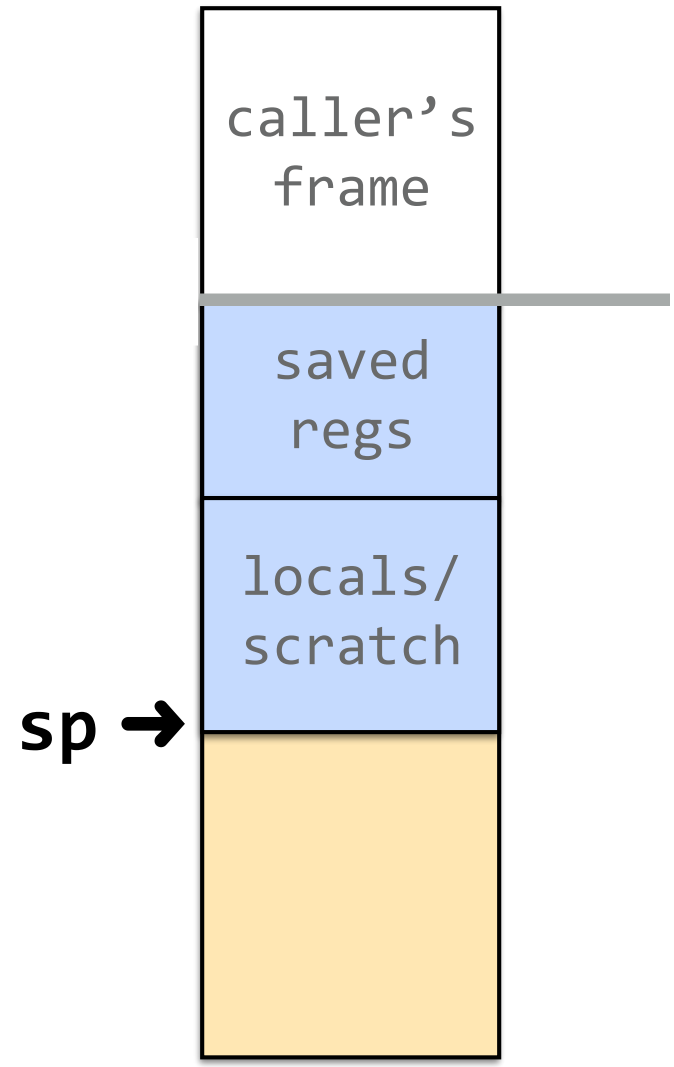


Diagram not to scale

Single stack frame

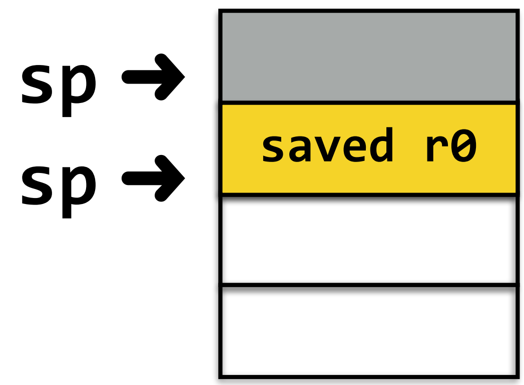
```
int winky(int a, int b)
{
   int c = 2*a;
   ...
   return c;
}
```



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

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

If winky calls binky…

<u>Why</u> do they collide on use of `lr` ?

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

What do we do about it?

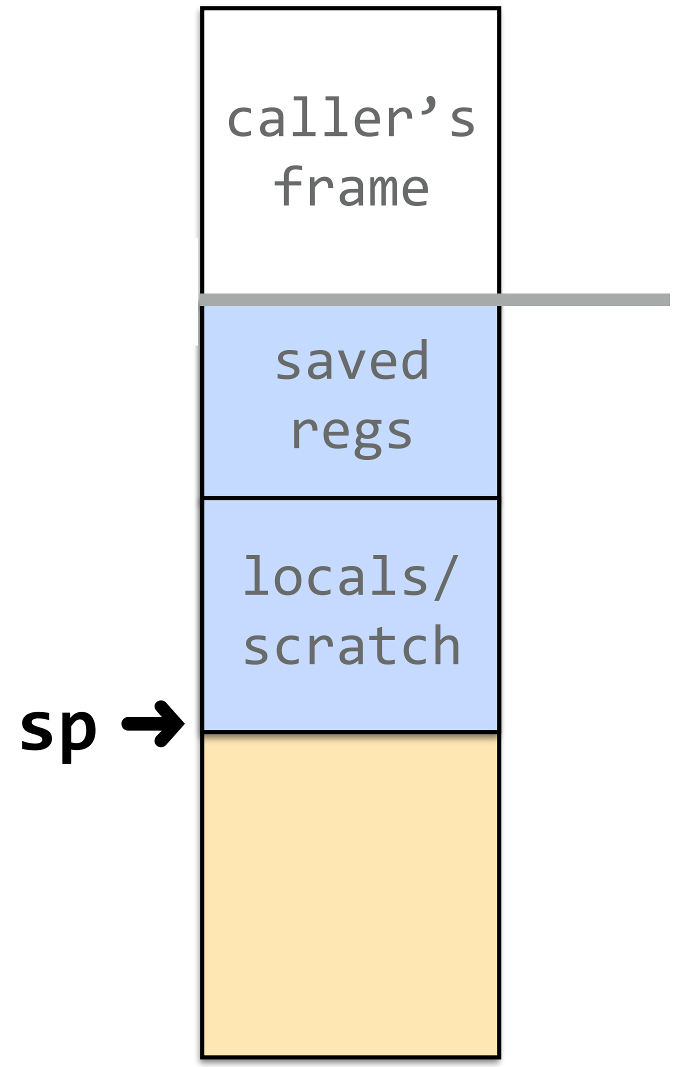
use stack as temp storage!

example.c

sp in constant motion

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

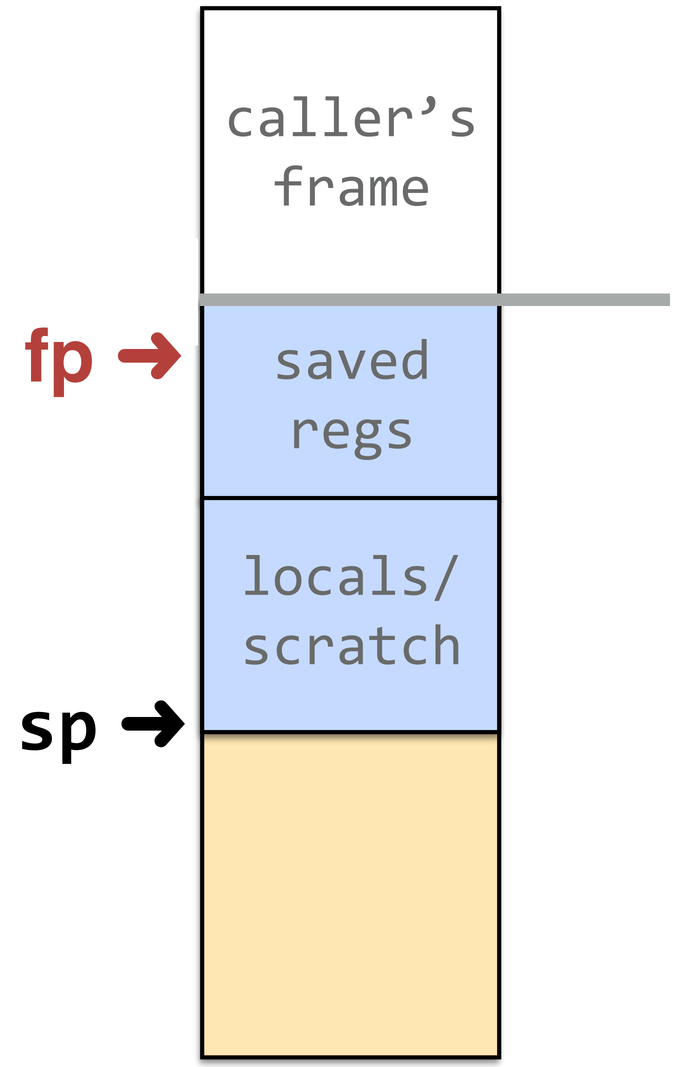
sp is constantly changing!
(push, pop, add sp, sub sp)



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

Prolog

`push fp, r13, lr, pc`

set `fp` to first word of stack frame

Body

`fp` stays anchored

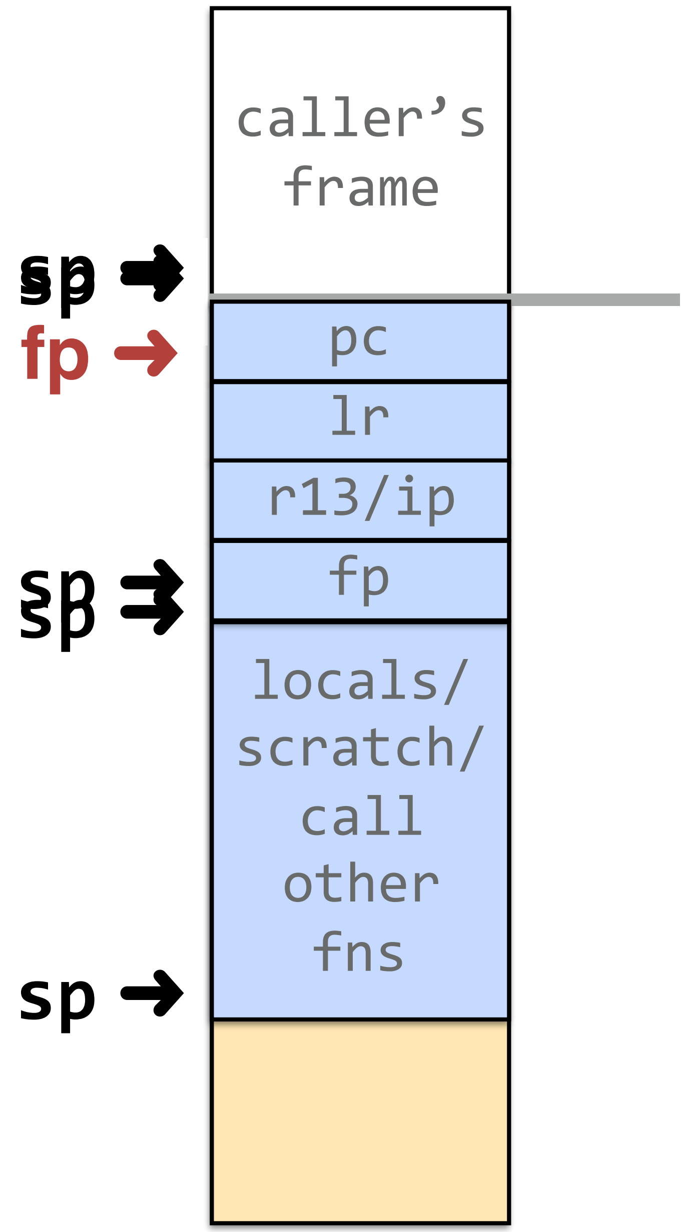
access data on stack `fp`-relative

offsets won't vary even if `sp` changing

Epilog

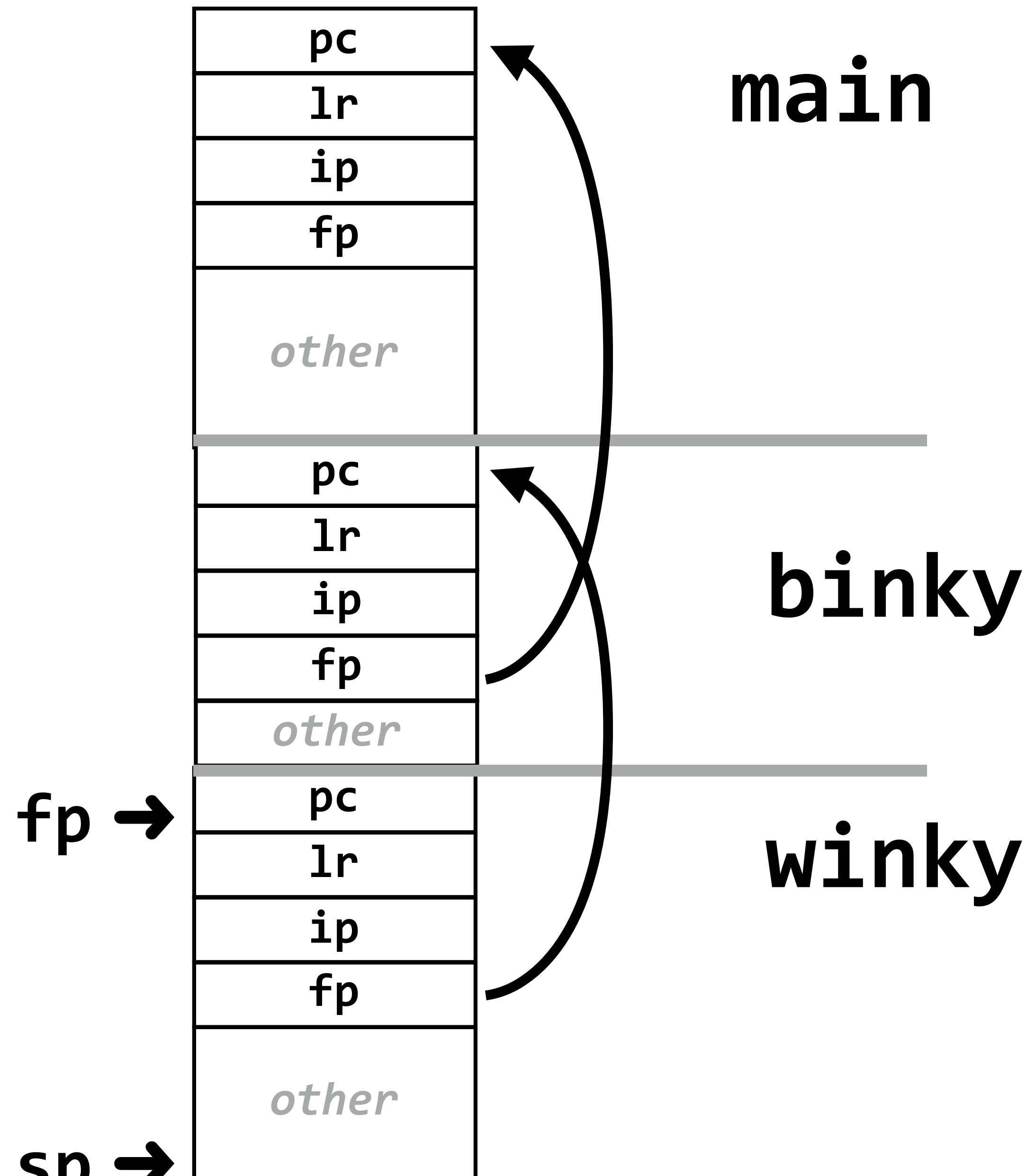
`pop fp, r13, lr`

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