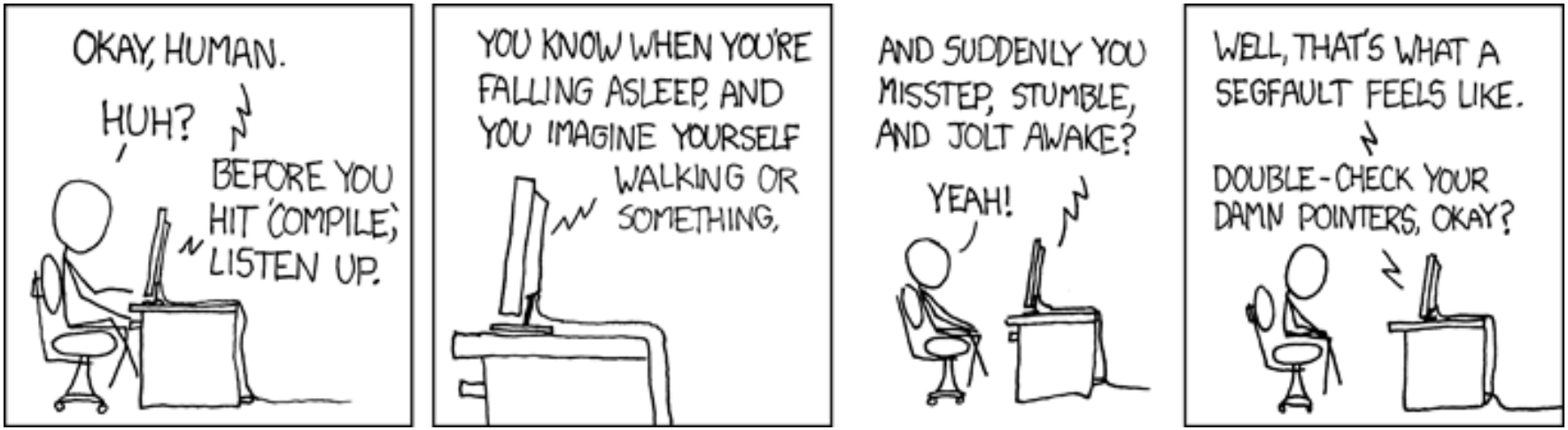
Topics for today

Mechanics of 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)

Accessing data by address/location is ubiquitous and powerful!

You already know that pointers are useful

- Sharing data instead of redundancy/copying
- Construct linked structures (lists, trees, graphs)
- Dynamic/runtime allocation

Now we learn how it works under the hood

- Memory-mapped peripherals located at fixed address
- Access to struct fields and array elements by relative location

`ldr/str` in assembly expressed in C as pointers — what do we gain?

- Type system adds readability, some safety
- Pointee and level of indirection now explicit in the type
- Access contiguous data at offset/relative position using pointer arithmetic

Address arithmetic

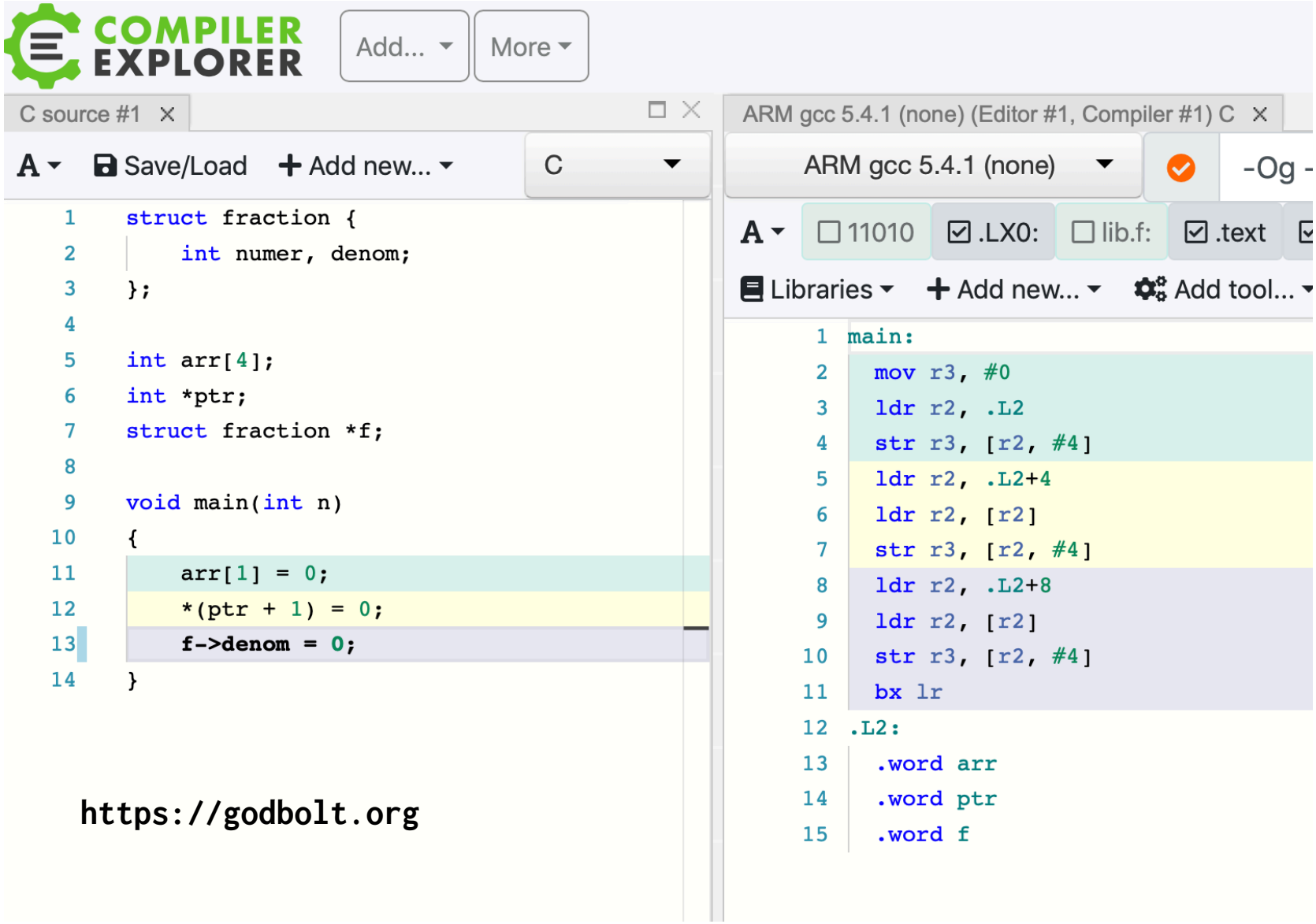
Manipulate address numerically
C pointer add/subtract is scaled by sizeof(pointee)

```
unsigned int *p = 0x20200000;
p = p + 1;          // p is now 0x20200004
```

Fancy ARM addressing modes

```
ldr r0, [r1, #4]           // constant displacement
ldr r0, [r1, r2]           // variable displacement
ldr r0, [r1, r2, asl #3]   // scaled index
```

How do these relate to accessing data structures in C?



Q: How does type of pointee influence the generated assembly?

Arrays and 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!
```

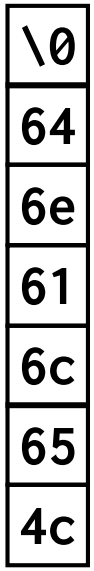
C-strings

No string "abstraction", just sequence of chars in memory, e.g. char array

`char *` points to first char of sequence, terminated by null (zero byte)

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

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



Excerpted from blink.s

```
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
```

```
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`

ARM architecture

no hosting OS

embedded ABI

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 `delta`

`delta` is callee of `main`

`delta` is caller 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 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 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 treated the same?

The stack to the rescue!

Configure region in memory to store data for executing functions(s)

Stack frame per function, has local variables, scratch space, saved register values

- 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, #0x8000000
bl main
```

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


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


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

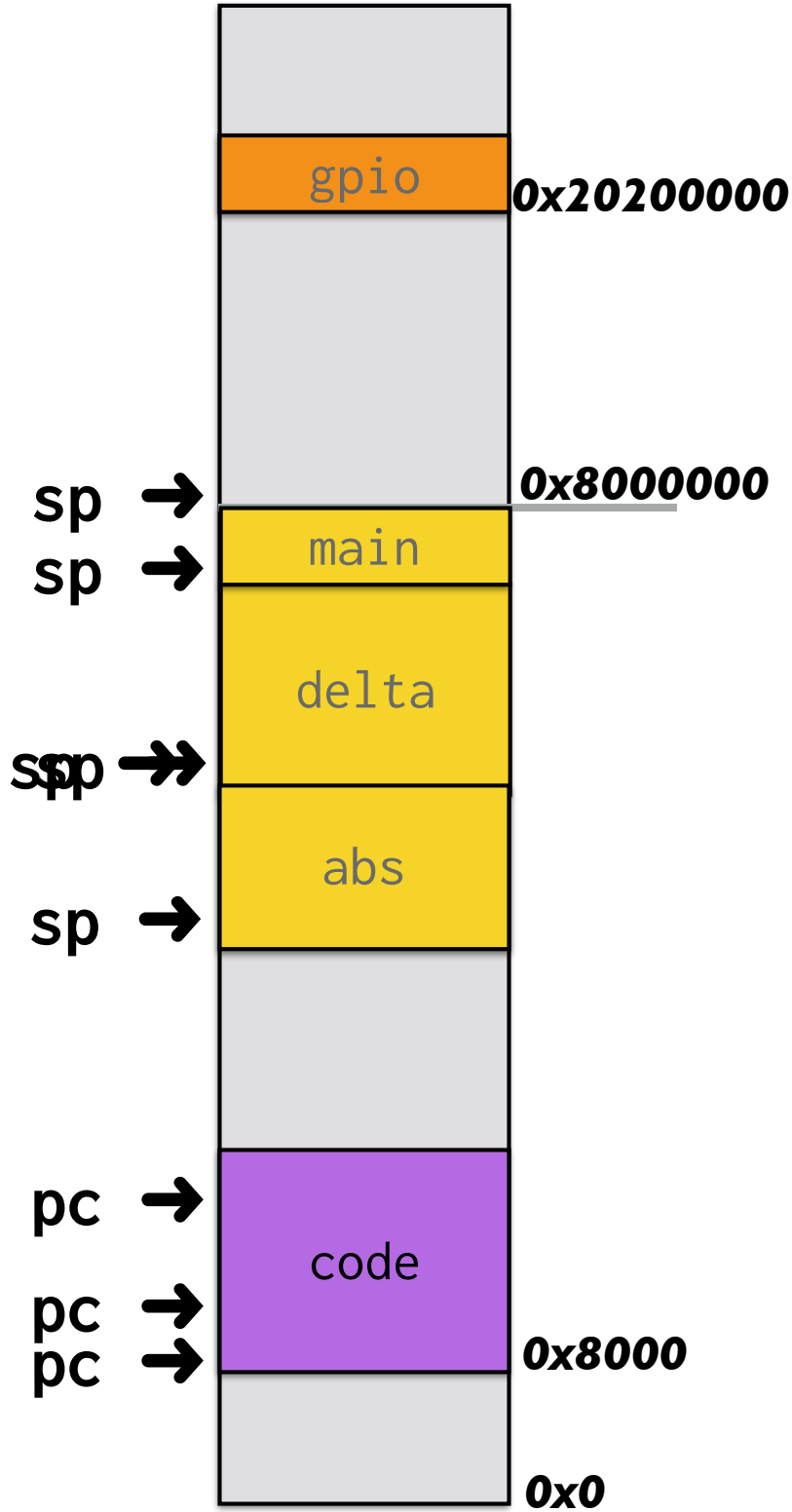
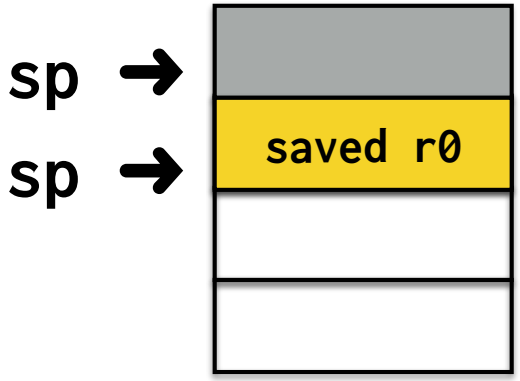


Diagram not to scale

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

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 on Pi, we have no such debugger 😢

But, **gdb** has simulation mode where it pretends to be an ARM processor, running on your laptop 🙌

Pretty good approximation (not perfect though, 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
```

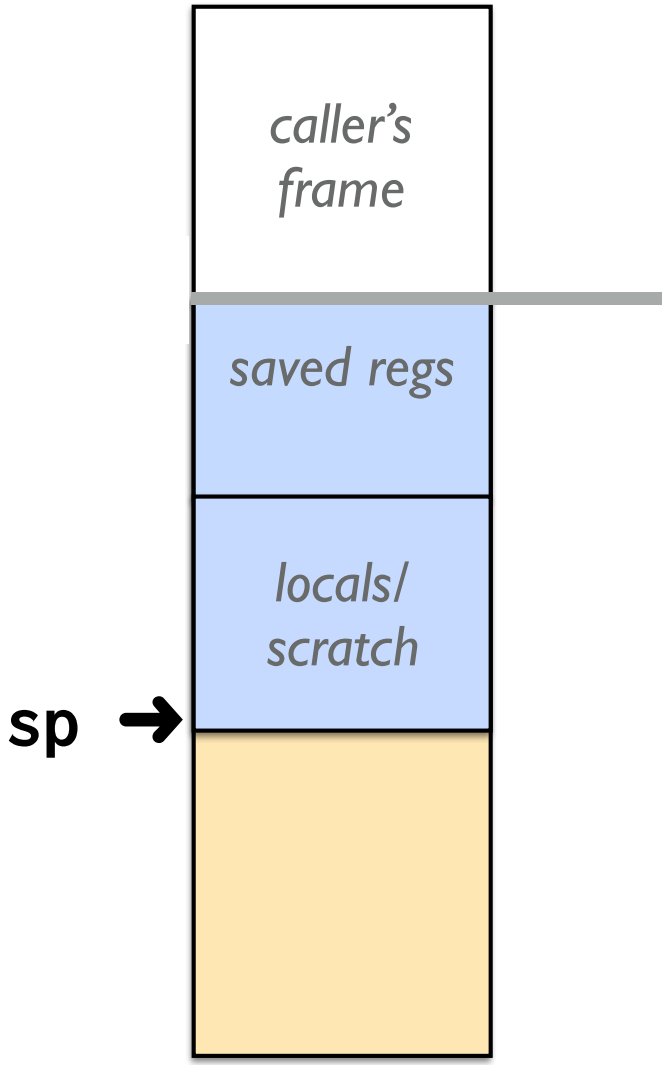
Read our guide to gdb simulation

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

sp in constant motion

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

sp is constantly changing!

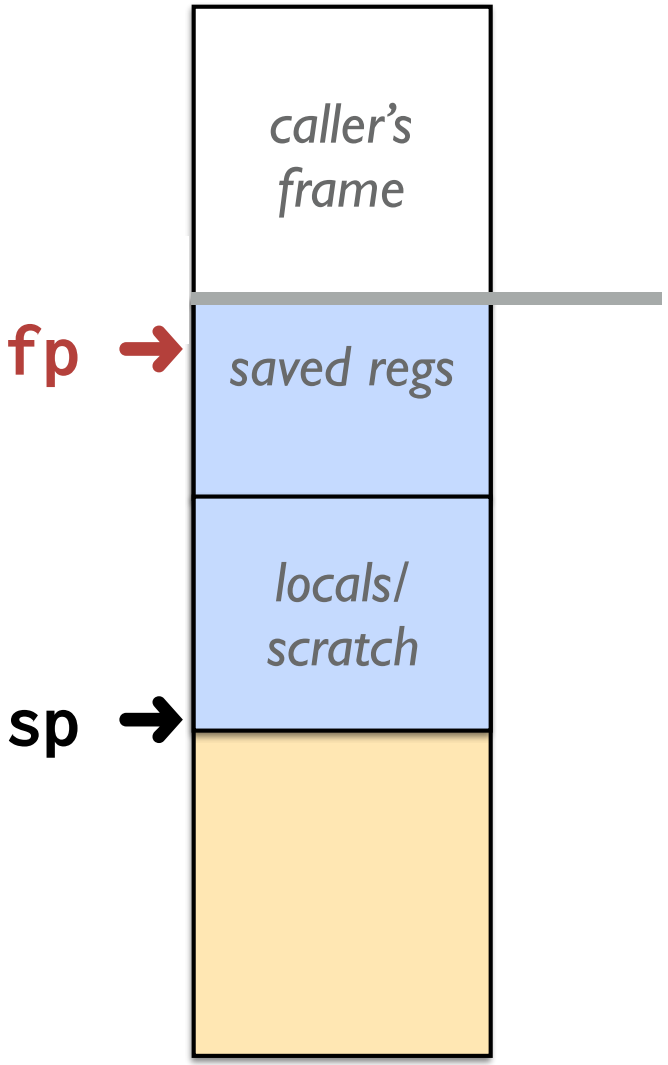


Add frame pointer

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

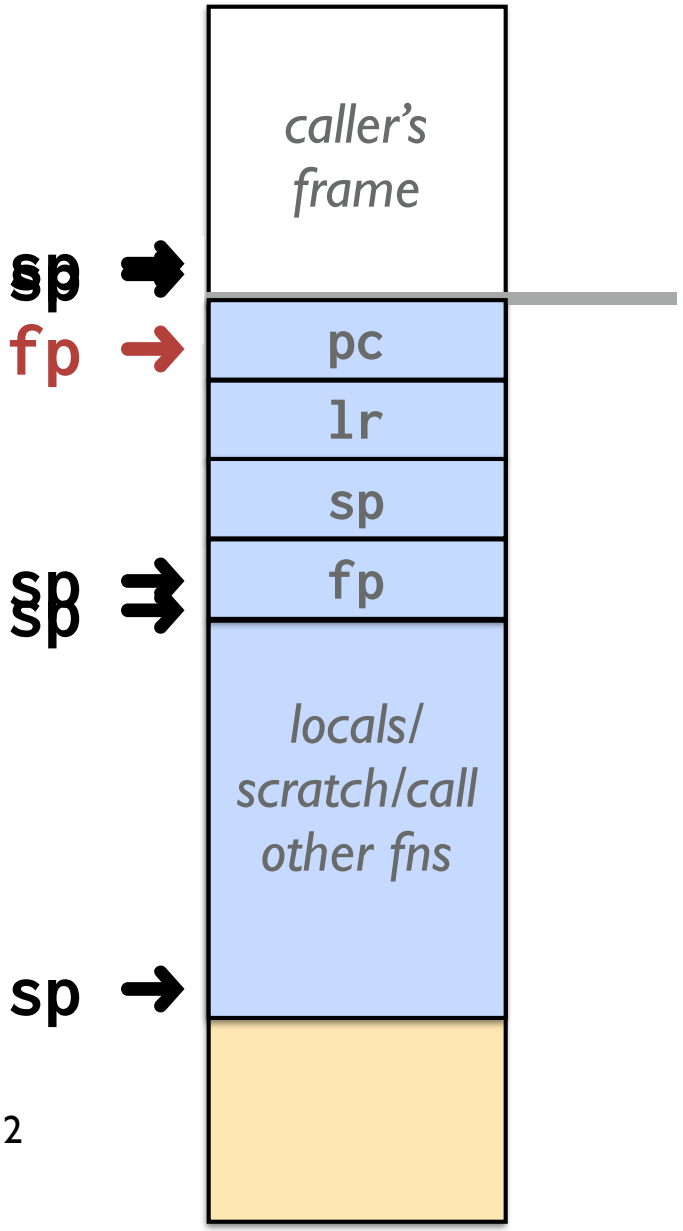
`fp` stays anchored

access data on stack `fp`-relative

offsets won't vary even if `sp` changing

Epilog

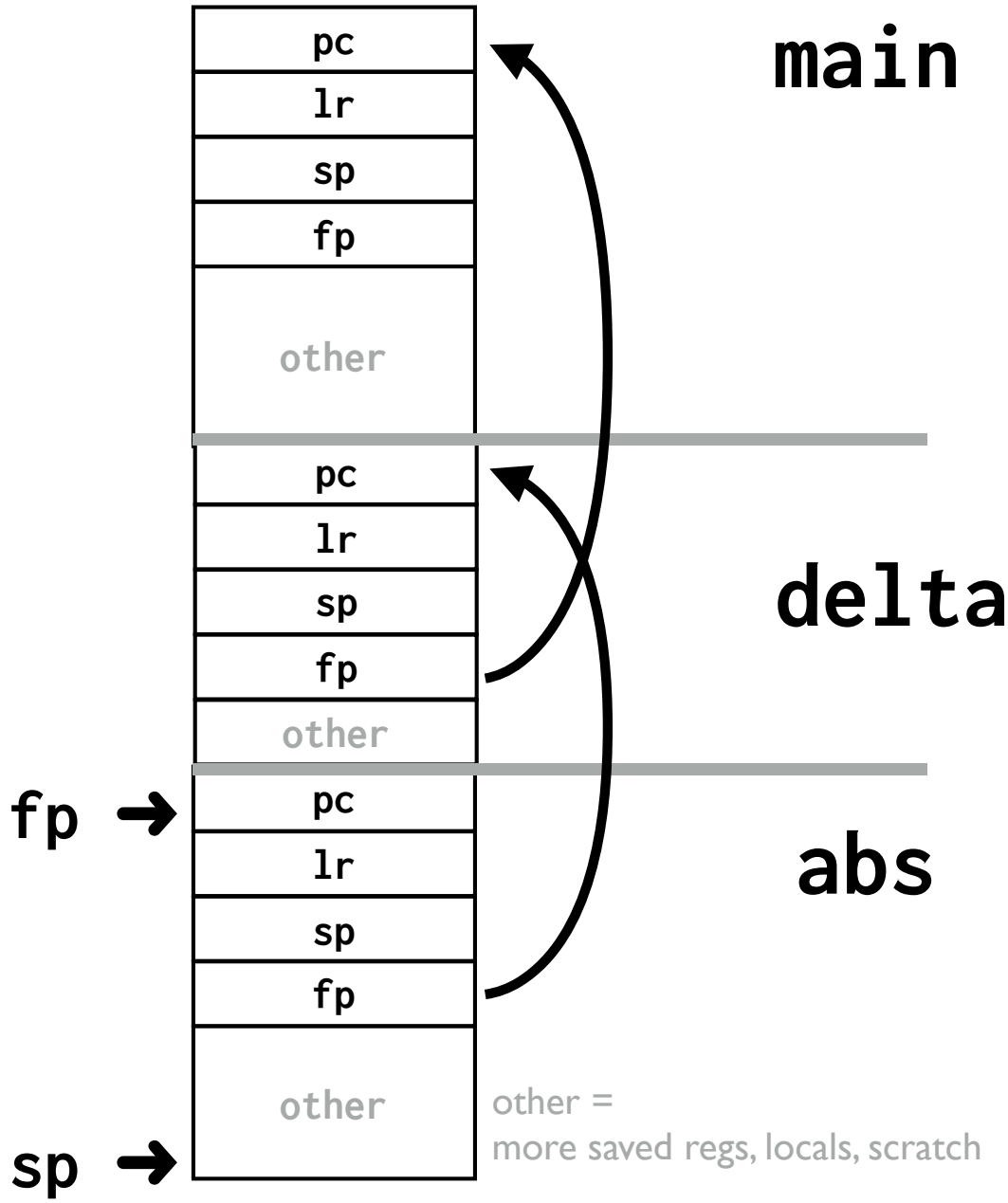
`pop fp, sp*, lr, pc*`



* 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 (**abs**) 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, #0x8000000
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

- Expensive, every function call affected
- Extra ~5 instructions to setup/tear down frame for each call
- 4 registers push/pop => extra 16 bytes per frame
- `fp` monopolizes use of one of our precious registers