Concepts So Far

ARM processor and memory architecture

ARM assembly language and machine code

Raspberry Pi peripherals and GPIO

C

- **Relationship between C statements and assembly language instructions**
- **Relationship between C types and allocation of data in memory (including pointers)**

Where Are We Going?

Functions (today)

Modules and libraries: combining files using the linker

Memory management: loading the program into memory and allocating memory while running

Communication: Serial protocol and ASCII, boot loader, printf

Bare metal stuff: cooler stuff from PAL and DE

Anatomy of C Function

```
int factorial(int n)
{
  int product = 1;
  for (int i = 2; i <= n; i++)
    product *= i;
  return product;
}
```

Call and return

Pass arguments

Return value

Local variables

Complication:

nested function calls

Call and Return

```
// excerpted from blink.s
loop:
   ldr r0, =SET0       // set pin high
   str r1, [r0]
   mov r2, #DELAY      // delay
   subs r2, #1
   bne .-4
   ldr r0, =CLR0       // set pin low
   str r1, [r0]
   mov r2, #DELAY      // delay
   subs r2, #1
   bne .-4
   b loop
```

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

delay:
  mov r2, #DELAY
  subs r2, #1
  bne .-4
 // but...
    how to return when loop finished?
```

```
    ldr r0, =SET0
    str r1, [r0]
    mov r14, pc  // note that pc=.+8
    b delay
    ldr r0, =CLR0
    str r1, [r0]
    mov r14, pc
    b delay
    b loop

delay:
    mov r2, #DELAY
    subs r2, #1
    bne .-4
    mov pc, r14
```

```
    ldr r0, =SET0
    str r1, [r0]
    mov r0, #DELAY
    mov r14, pc
    b delay
    ldr r0, =CLR0
    str r1, [r0]
    mov r0, #DELAY
    mov r14, pc
    b delay
    b loop

delay:
    subs r0, #1
    bne .-4
    mov pc, r14
```

```
// Call and Return

// b - branch

// New variations of branch

// branch and link (call)
bl delay // pc=delay, lr=.+4 (=pc-4)


// branch to address in reg lr (return)
bx lr    // pc=lr
```

Arguments and Return Values

```
…
mov r0, #DELAY
b delay
…

// Conventions

r0-r3 used to pass values as arguments
r0-r1 used to return values
```

ABI

ABI = Application Binary Interface

Conventions for calling functions

Allows different languages to interoperate

ARM uses extended-ABI (eabi) as in `arm-none-eabi`

Callers and Callees

Nomenclature

- *caller* - function doing the calling
- *callee* - function called

Register Ownership

`r0-r3` **callee-owned registers**

- **Callee can change these registers**
- **Caller can not assume the values are the same as when the callee was called**

`r4-r13` **are caller-owned registers**

- **Callee should not change these registers**
- **They should be the same when the function returns; callers assumes they are the same**

Discuss

1. What does the callee need to do if it wants to use a register that the caller owns?

2. What is the advantage of making most registers caller-owned?

3. What happens to argument `r0` passed by the caller to the callee, if the callee calls another function that has an argument (`r0`)?

Callee-Saved

1. The callee saves registers only if the callee needs to use them; if the callee does not use them, there is no need to save them

More efficient than having the caller save and restores all these registers

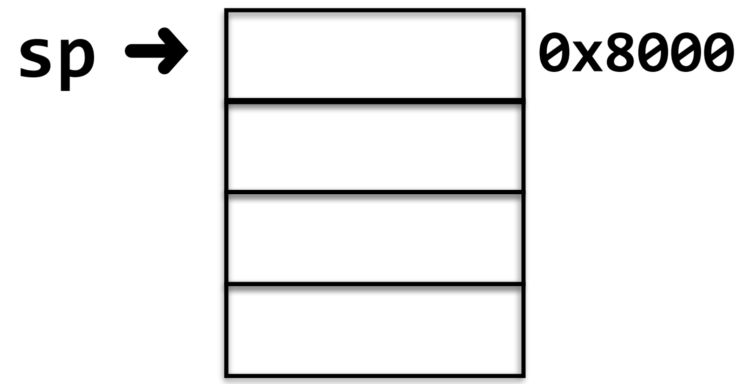
2. `r0` will be overwritten when passing an argument; if callee still needs it, should save it

Where does the callee save registers?

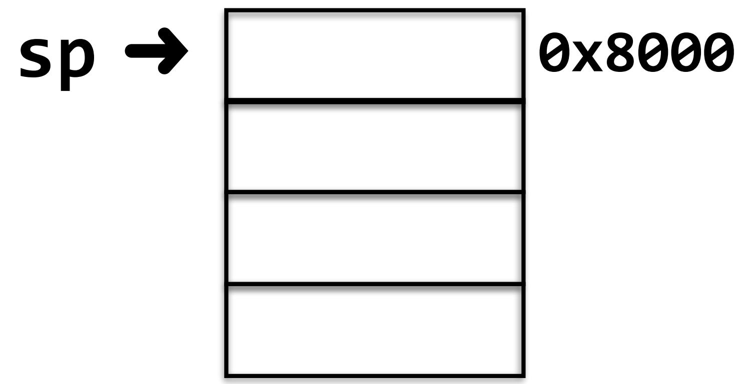
Saving Registers

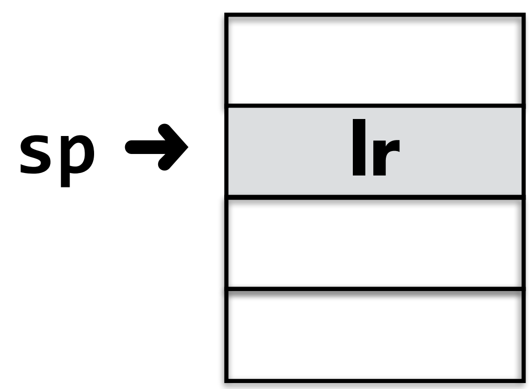
The Stack to the Rescue

```
// start.s
// unsigned *sp = 0x8000;
mov sp, #0x8000
```

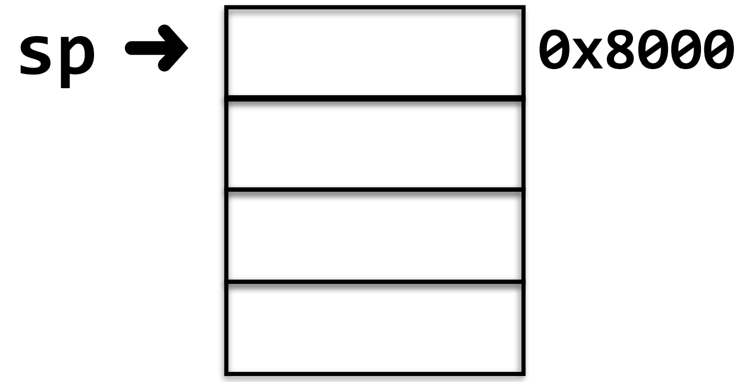


```
// start.s
// unsigned *sp = 0x8000;
mov sp, #0x8000

// push
// *--sp = lr
// decrement sp before store
str lr, [sp, #-4]!
push {lr}
```

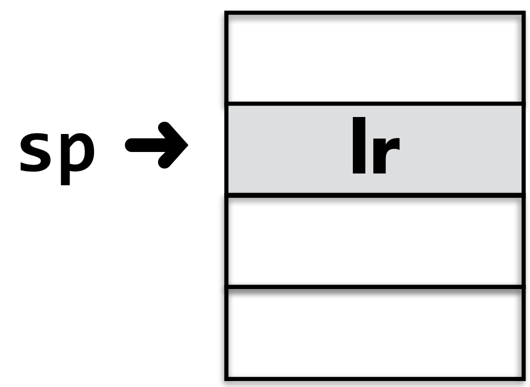




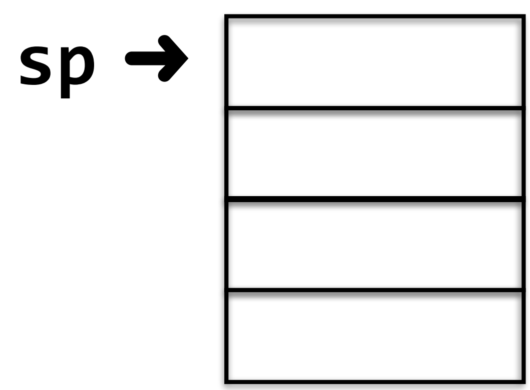
```
// start.s
// unsigned *sp = 0x8000;
mov sp, #0x8000
```

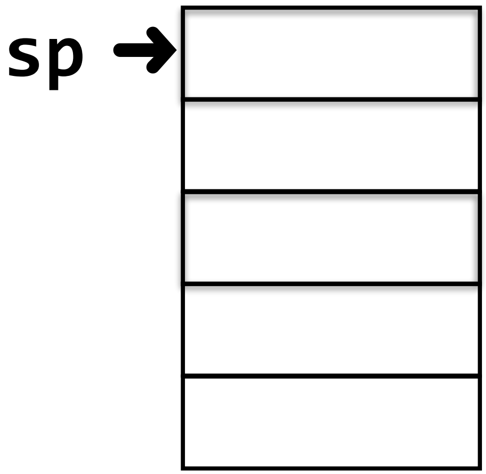



```
// push
// *--sp = lr
// decrement sp before store
str lr, [sp, #-4]!
push {lr}
```



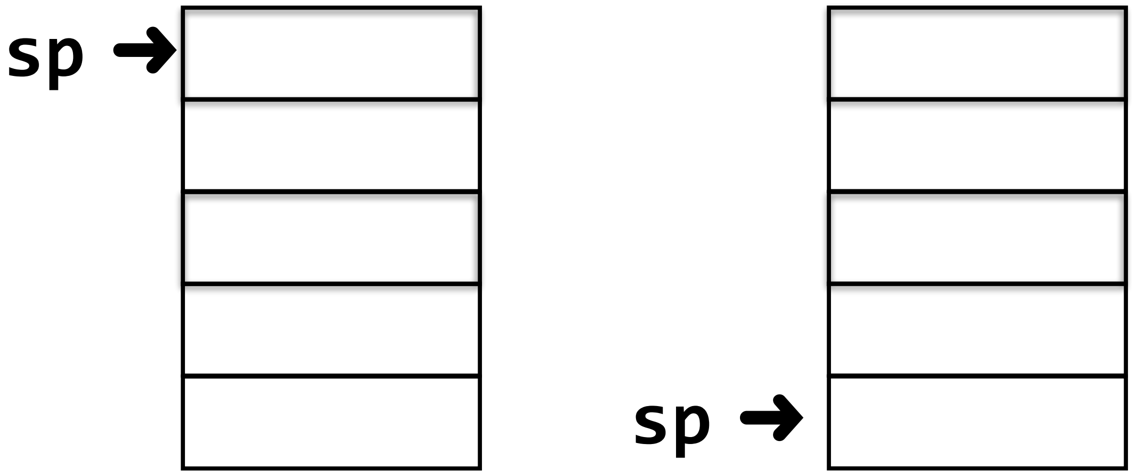

```
// pop
// lr = *sp++
// increment sp after load
pop {lr}
ldr lr, [sp], #4
```






```c
int binky(int x)
{
  int y = x + 2;
  return y;
}
```

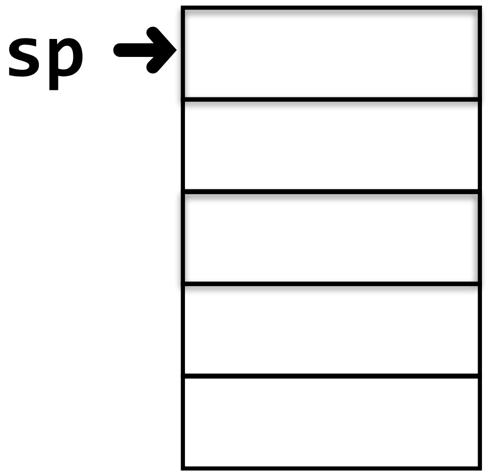
```
binky:
  sub sp, sp, #16
  str r0, [sp, #4]
  ldr r3, [sp, #4]
  add r3, r3, #2
  str r3, [sp, #12]
  ldr r3, [sp, #12]
  mov r0, r3
  add sp, sp, #16
  bx  lr
```

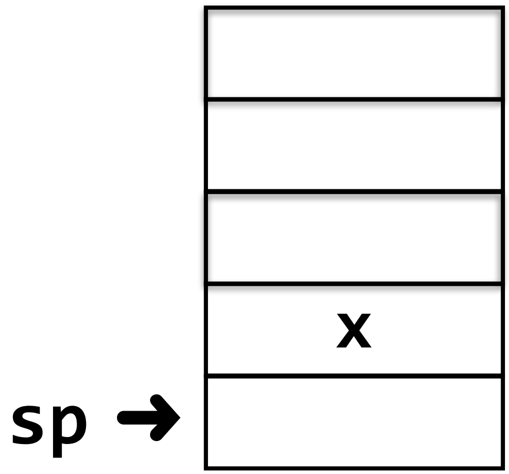


EABI requires `sp` to be *aligned* to a multiple of `8`

```
int binky(int x)
{
  int y = x + 2;
  return y;
}
```

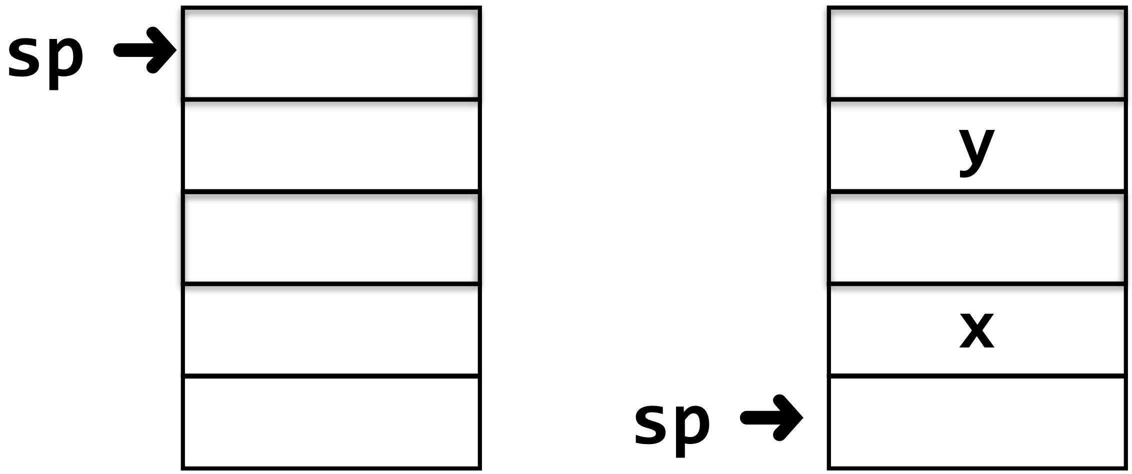
```
binky:
  sub sp, sp, #16
  str r0, [sp, #4]
  ldr r3, [sp, #4]
  add r3, r3, #2
  str r3, [sp, #12]
  ldr r3, [sp, #12]
  mov r0, r3
  add sp, sp, #16
  bx  lr
```





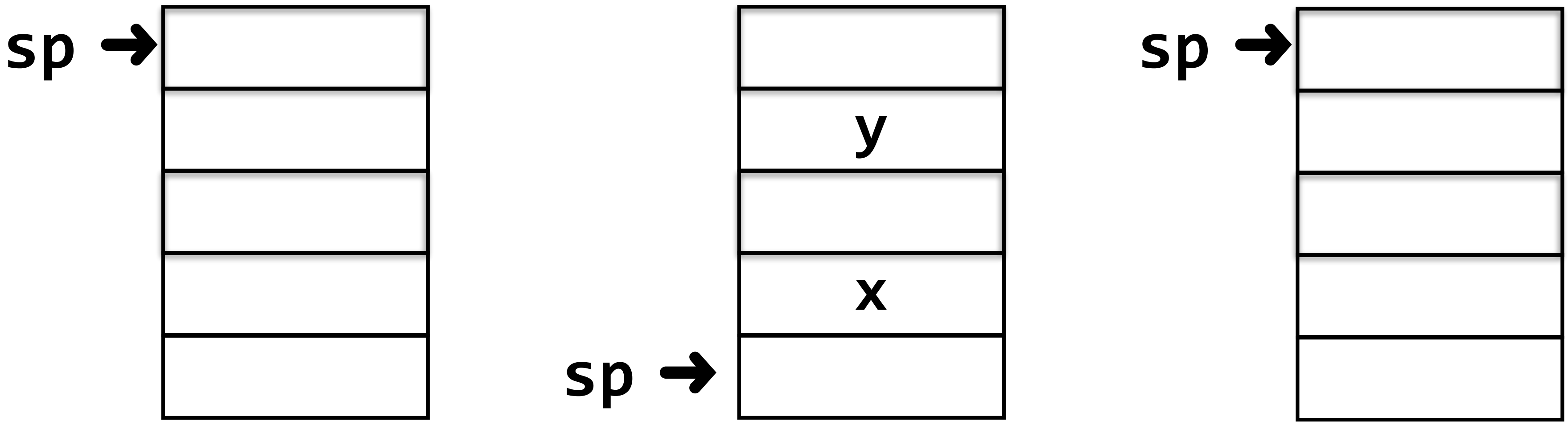

```
int binky(int x)
{
  int y = x + 2;
  return y;
}
```

```
binky:
  sub sp, sp, #16
  str r0, [sp, #4]
  ldr r3, [sp, #4]
  add r3, r3, #2
  str r3, [sp, #12]
  ldr r3, [sp, #12]
  mov r0, r3
  add sp, sp, #16
  bx  lr
```




```
int binky(int x)
{
  int y = x + 2;
  return y;
}
```

```
binky:
  sub sp, sp, #16
  str r0, [sp, #4]
  ldr r3, [sp, #4]
  add r3, r3, #2
  str r3, [sp, #12]
  ldr r3, [sp, #12]
  mov r0, r3
  add sp, sp, #16
  bx  lr
```

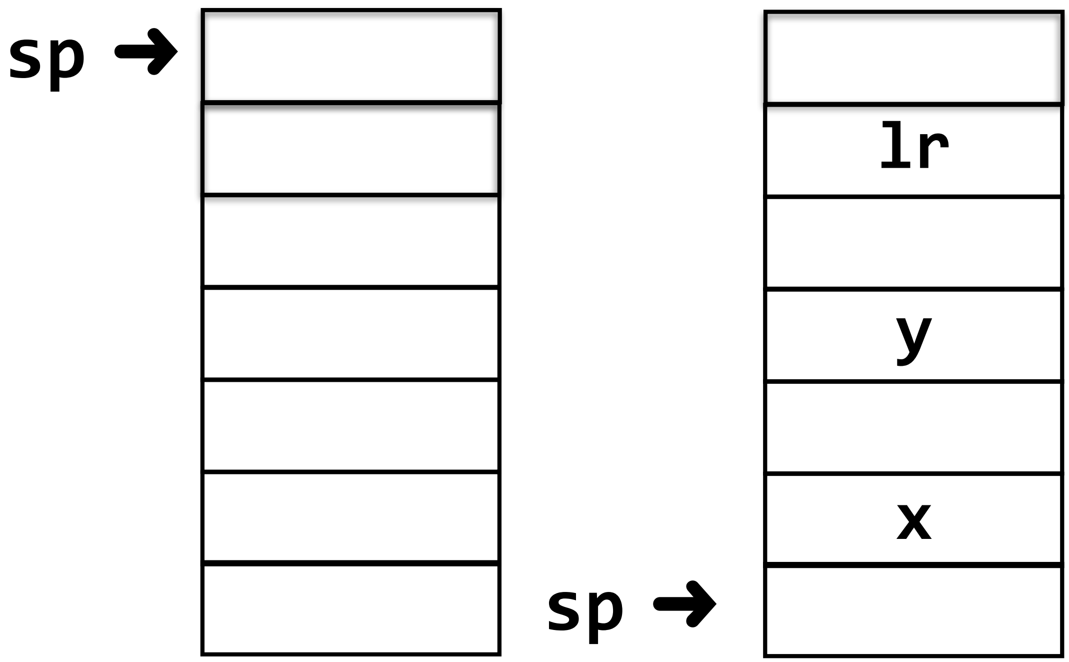



```
int binky(int x)
{
  int y = x + 2;
  return y;
}
```

```
binky:
  sub sp, sp, #16
  str r0, [sp, #4]
  ldr r3, [sp, #4]
  add r3, r3, #2
  str r3, [sp, #12]
  ldr r3, [sp, #12]
  mov r0, r3
  add sp, sp, #16
  bx  lr
```

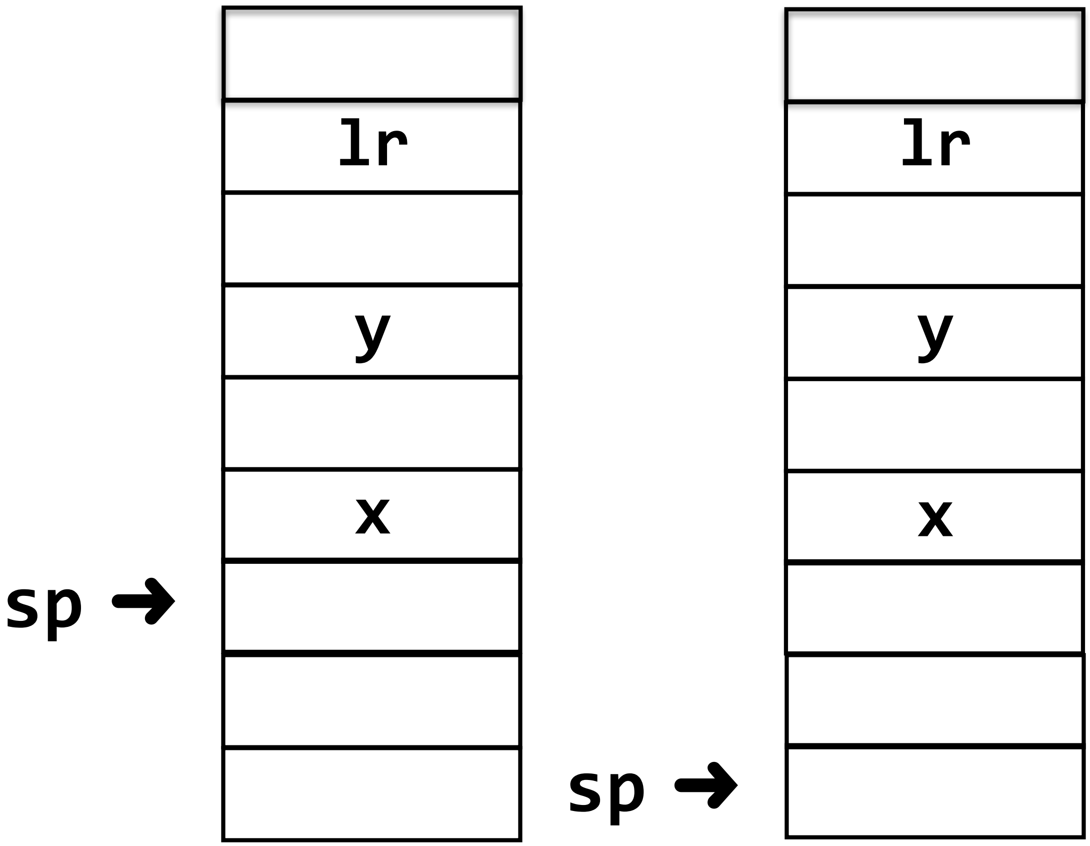
```
// call binky inside winky

int winky(int x)
{
  int y = binky(x+1);
  return x+y;
}

// What happens to lr
//   when you call binky?
```

Problem: lr changes



Solution: push/pop lr

```
winky:
   push {lr}
   sub sp, sp, #20
   str r0, [sp, #4]
   ldr r3, [sp, #4]
   add r3, r3, #1
   mov r0, r3
   bl  binky
   str r0, [sp, #12]
   ldr r2, [sp, #4]
   ldr r3, [sp, #12]
   add r3, r2, r3
   mov r0, r3
   add sp, sp, #20
   pop {lr}
   bx  lr
```



What if sp changes?

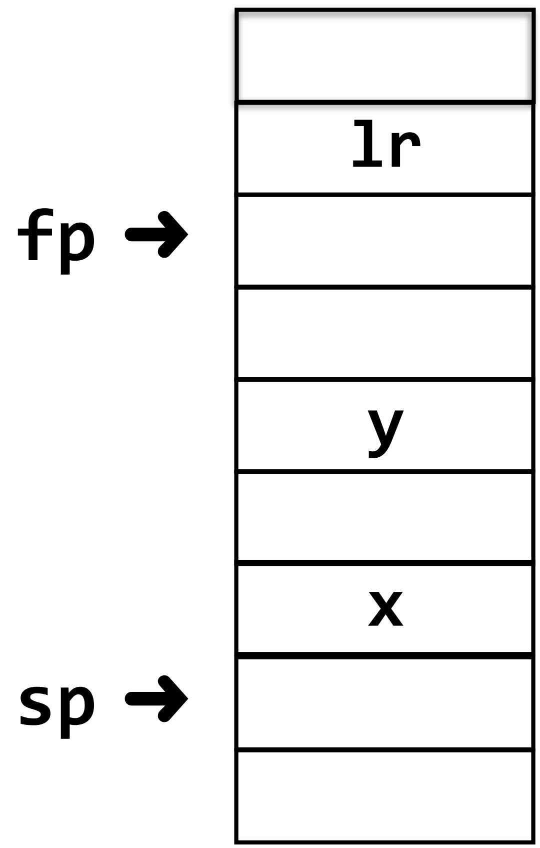
```
winky:
   push {lr}
   sub sp, sp, #20
   str r0, [sp, #4]
   ldr r3, [sp, #4]
   add r3, r3, #1
   mov r0, r3
   bl  binky
   str r0, [sp, #12]
   ldr r2, [sp, #4]
   ldr r3, [sp, #12]
   add r3, r2, r3
   mov r0, r3
   add sp, sp, #20
   pop {lr}
   bx  lr
```

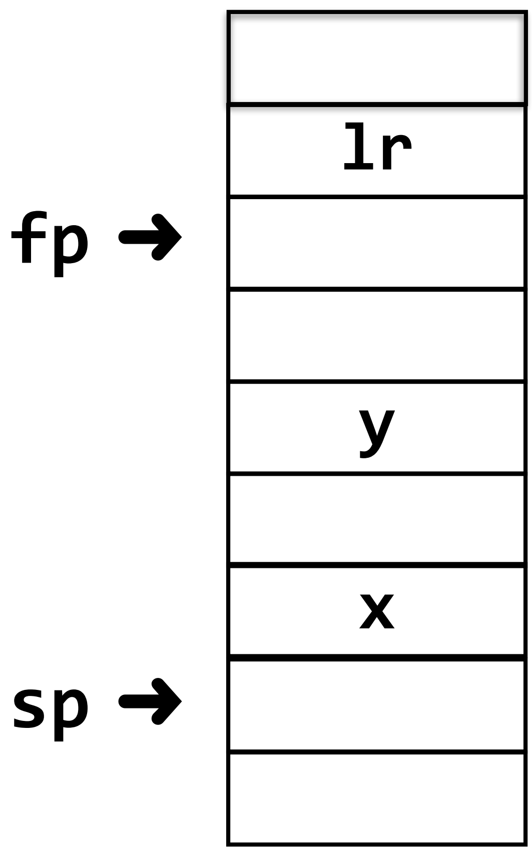
The Frame Pointer (FP)

Establish a stack *frame*

Provides "anchor" for the local variables used by a function

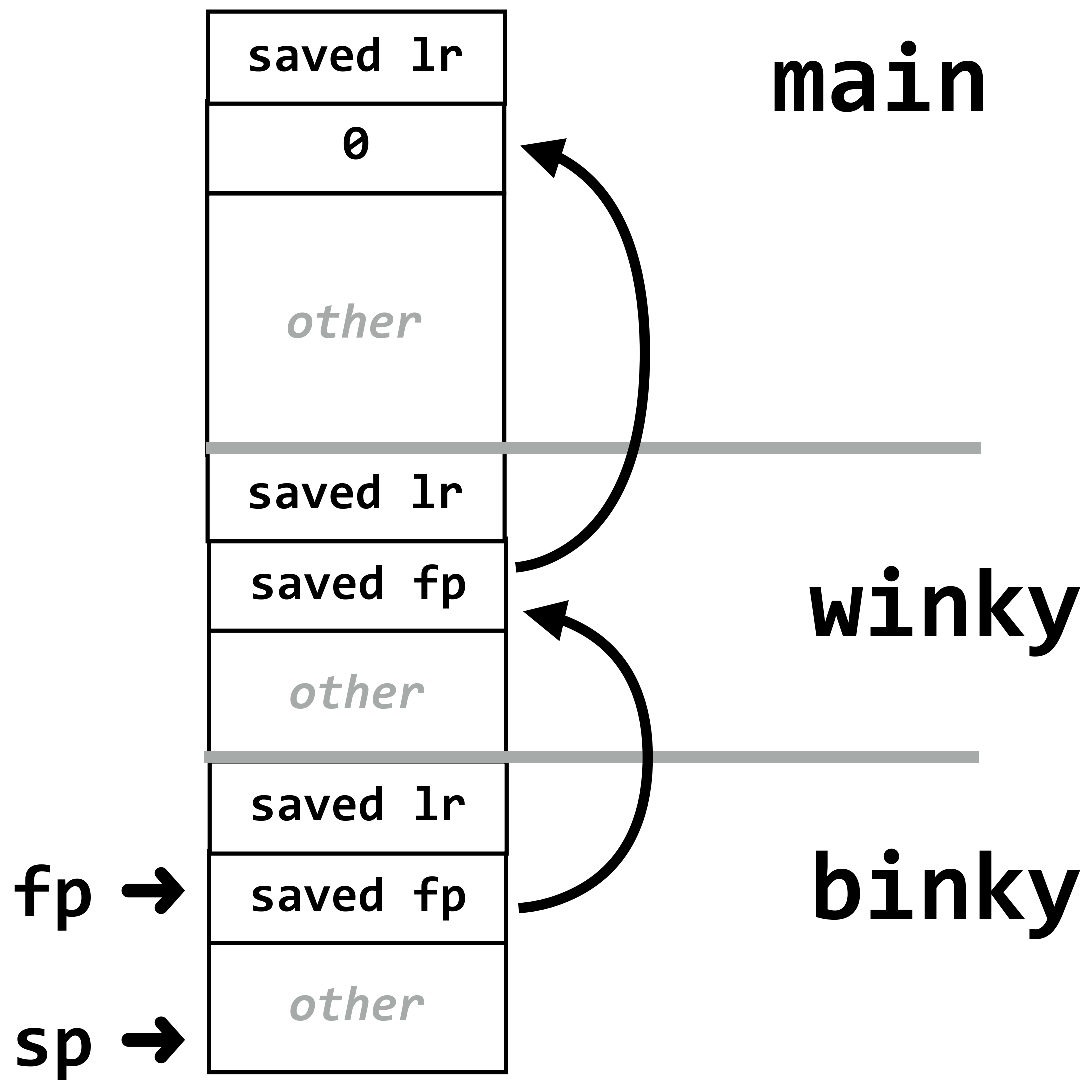
Locals offset relative to `fp`

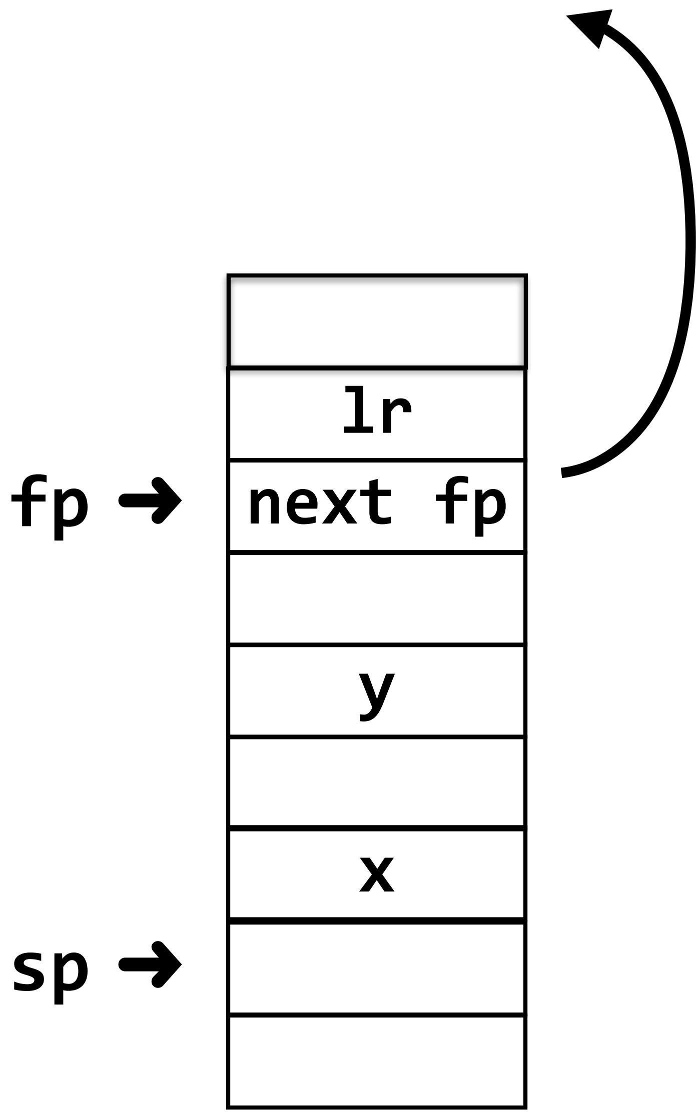





```
winky:
   …
   str r0, [fp,#-16]
   ldr r3, [fp,#-16]
   add r3, r3, #1
   mov r0, r3
   bl  binky
   str r0, [fp,#-8]
   ldr r2, [fp,#-16]
   ldr r3, [fp,#-8]
   add r3, r2, r3
   mov r0, r3
   …
   bx  lr
```

Create Linked List of Function Stack Frames






```
winky:
   push {fp, lr}
   add fp, sp, #4
   sub sp, sp, #16

   …

   sub sp, fp, #4
   pop {fp, lr}
   bx  lr
```

```
// start.s

// Need to initialize fp = NULL

.globl _start
_start:
    mov sp, #0x8000
    mov fp, #0 // fp=NULL
    bl main
hang:
    b hang
```

The Frame Pointer (FP)

Establishes a stack frame

Enables:

- **Locals offset relative to `fp` (not `sp`, since `sp` may change)**
- **Backtrace for debugging**
- **Unwind stack on exception**

Special Registers

`r4-r15` caller-owned, callee-save registers

`r11 (fp)` - special

`r12 (ip)` - special (scratch register)

`r13 (sp)` - special

`r14 (lr)` - special

`r15 (pc)` - special

Summary

Calling functions:

- **link register (`lr`)**
- **ABI: arguments and return values**

The stack and stack pointer (`sp`)

Stack frames and the frame pointer (`fp`)

Extra

ARM Addressing Modes

```
  str r0, [r1]                // indirect

Preindex, non-updating
  str r0, [r1, #4]            // constant displacement
  str r0, [r1, r2]            // variable displacement
  str r0, [r1, r2, lsl #4]  // scaled index

Preindex, writeback (update before use)
  str r0, [r1, #4]!           // r1 pre-updated += 4
  str r0, [r1, r2]!
  str r0, [r1, r2, lsl #4]!

Postindex (update after use)
  str r0, [r1], #4            // r1 post-updated += 4
  ...
```