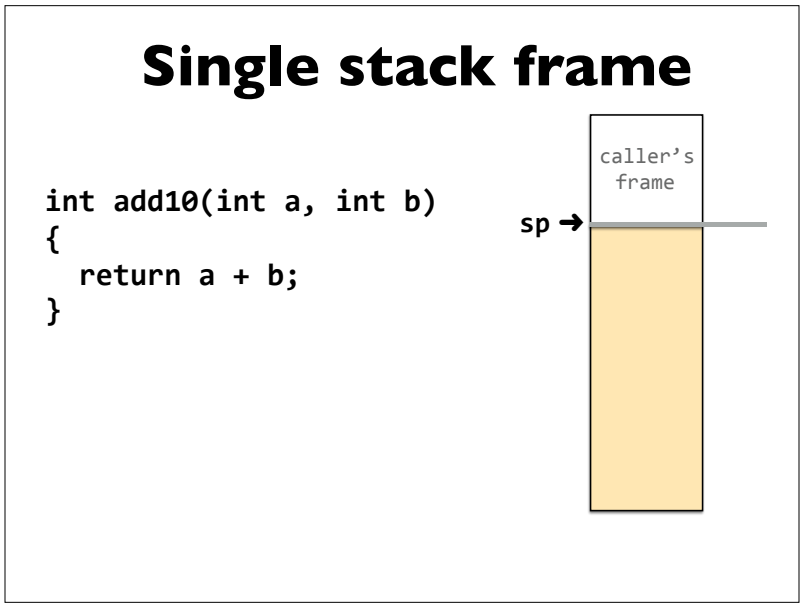
Stack Frames



The stack is a block of memory reserved for a program. Stacks usually go *down* -- the start of the stack is its highest address. As the program executes, the stack grows down as more space is needed, and then shrinks up as that space can be reclaimed.

The primary function a stack serves is to store temporary state of functions. When the compiler takes your source code and turns it into machine instructions, one of the things it does is add instructions to manage the stack. When a function is called, it stores any state that it needs to restore on the stack. Exactly how much state this is depends on the function. For example, a function like

```
int add(int x, int y) {
  return x + y;
}
```

doesn't need to store any state. The parameters x and y are passed in r0 and r1, and it returns the result in r0. So all this function needs to do is add r0 and r1, store the result in r0, and return. Since the return address it should go to is store in the link register lr, all it needs to do is branch back to lr. If you compile this function with arm-none-eabi-gcc with -Og, the assembly is:

```
add:
        add     r0, r0, r1
        bx      lr
```

More complicated functions, though, that allocate a lot of variables, large variables, or arrays, need space for them. This space is on the stack.

Generally speaking, compilers try to put variables in registers. Registers are fast. But recall that of the 16 registers, only r0-r3 are callee-owned. If a function wants to modify r4, it needs to make sure that it restores r4 to what it was before it returns. Therefore, non-trivial functions, when they're called, usually save all of the caller-owned registers they're going to use. E.g., r4.

One register that's very commonly saved is lr. Recall that the link register stores the address of the instruction the function should branch to when it returns. This is automatically populated with the bl (branch-and-link) instruction. If a function calls any other function, then lr will be overwritten. It

therefore needs to be saved. If you compile this add10 function

```
int add10(int a) {
  return add(10, a);
}
```

this is the generated assembly:

```
add10:
        stmfd   sp!, {r4, lr}
        mov     r1, r0
        mov     r0, #10
        bl      add
        ldmfd   sp!, {r4, lr}
        bx      lr
```

here is equivalent assembly from gcc v. 8.2:

```
add10:
        push    {r4, lr}
        mov     r1, r0
        mov     r0, #10
        bl      add
        pop     {r4, pc}
```

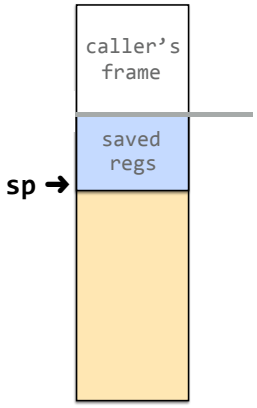
Note how `strmfd` and `ldmfd` have been replaced by `push` and `pop`: these are just useful shorthand and are equivalent. The

```
        stmfd   sp!, {r4, lr}
```

instruction pushes `r4` and `l4` onto the stack, by storing them at the memory pointed to by `sp`. The `!` after `sp` means that `sp` should be automatically decremented by the amount written. So after `strmfd` executes, `sp` will have gone down by 8 (since the stack grows down). Similarly, after the `ldrmfd` instruction the value of `sp` will go up by 8. The '`fd`' means 'full descending stack', so the stack grows down.

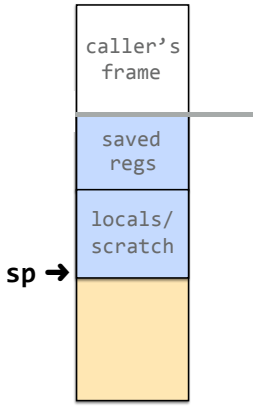
Single stack frame

```
int add10(int a, int b)
{
  return a + b;
}
```



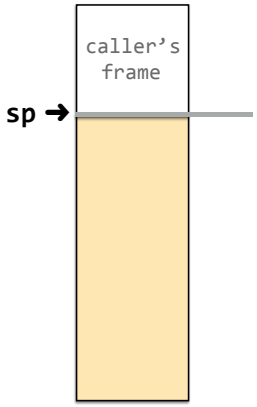
Single stack frame

```
int add10(int a, int b)
{
  return a + b;
}
```



Single stack frame

```
int add10(int a, int b)
{
  return a + b;
}
```



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 prologue/epilogue to each function that sets up/ tears down the standard frame and manages fp

You'll use this in assignment 4

The stack pointer is always moving -- up, down. For example, suppose you have a snippet of code like this:

```
if (x > y) {
    int array[100];
    // do some processing with array
} else {
  return x + y;
}
```

If the code takes the if branch, it will grow the stack by 400 bytes (100 words), then restore it when the code between the braces {} completes.

This means that, if you stop at any point in the code, it can be very hard to figure out exactly where the current stack frame starts. The compiler carefully generates code that will restore the stack properly, but this means figuring out exactly where the frame starts might require simulating the future instructions. Put another way, optimized code isn't intended to be read and easily understood - it's intended to be executed fast.

This can be tough if you want to run your code in a debugger. It's not hard to figure out what function you're in -- just look at the program counter and see where in the program that instruction is. But if you want to see the entire stack trace, you need to figure out what function called the current one. This requires knowing where the current stack frame starts, and so where the caller's stack frame ends.

To make this easier, ARM has defined a standard way, called ACPS, that a program manipulates stack frames. In addition to the stack pointer, ACPS maintains a "frame pointer." ACPS adds a prologue and epilogue to every function. The prologue sets up a standard format for the start of the stack frame, and stores the frame pointer. The frame pointer points to the beginning of the frame. The prologue always copies the frame pointer and link register to the stack, at a known offset in the stack frame. This means that if you have the frame pointer, you can find both the instruction this function will return to (so the calling function) as well as the frame pointer of the previous stack frame, so where that frame is stored.

Trace APCS

Prolog

`stmfd fp, sp, lr, pc`

set `fp` to first word of stack frame

Body

`fp` stays anchored

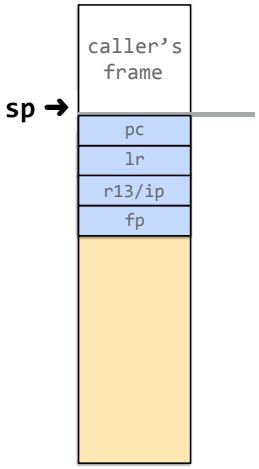
access data on stack `fp`-relative

offsets won't vary even if `sp` changing

Epilog

`ldmia fp, sp, lr`

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



This slide lays out precisely how APCS works. The prolog stores the frame pointer, stack pointer, lr, and program counter. It then sets the frame pointer to the first word of the stack frame.

While the function executes, the frame pointer stays put.

Then, before the function exits, it runs the epilogue. The epilogue returns the frame pointer, stack pointer, and link register. It then branches back to the link register, so the caller function.

You can't pop the pc from the stack because the pc stored on the stack is from the start of the function.

Trace APCS

Prolog

`stmfd fp, sp, lr, pc`

set `fp` to first word of stack frame

Body

`fp` stays anchored

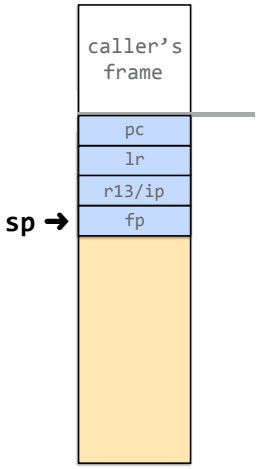
access data on stack `fp`-relative

offsets won't vary even if `sp` changing

Epilog

`ldmia fp, sp, lr`

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



Trace APCS

Prolog

`stmfd fp, sp, lr, pc`

set `fp` to first word of stack frame

Body

`fp` stays anchored

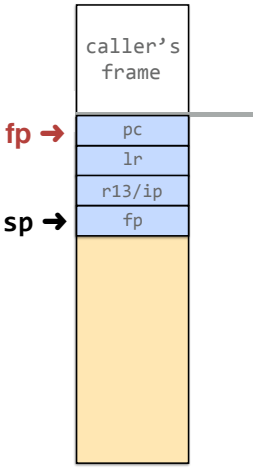
access data on stack `fp`-relative

offsets won't vary even if `sp` changing

Epilog

`ldmia fp, sp, lr`

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



Trace APCS

Prolog

`stmfd fp, sp, lr, pc`

set `fp` to first word of stack frame

Body

`fp` stays anchored

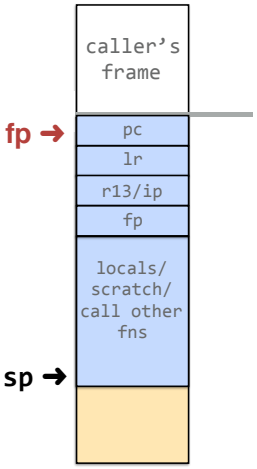
access data on stack `fp`-relative

offsets won't vary even if `sp` changing

Epilog

`ldmia fp, sp, lr`

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



Trace APCS

Prolog

`stmfd fp, sp, lr, pc`

set `fp` to first word of stack frame

Body

`fp` stays anchored

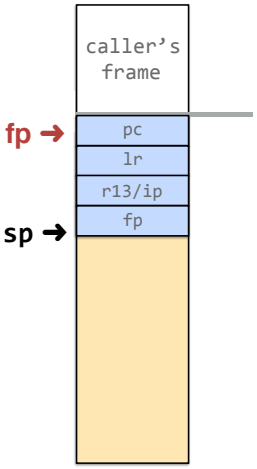
access data on stack `fp`-relative

offsets won't vary even if `sp` changing

Epilog

`ldmia fp, sp, lr`

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



Trace APCS

Prolog

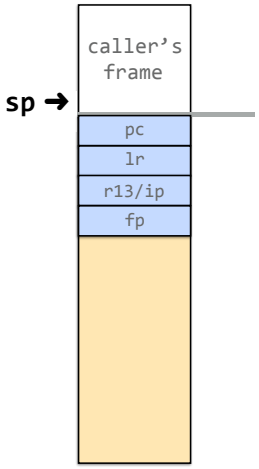
`stmfd fp, sp, lr, pc`
set `fp` to first word of stack frame

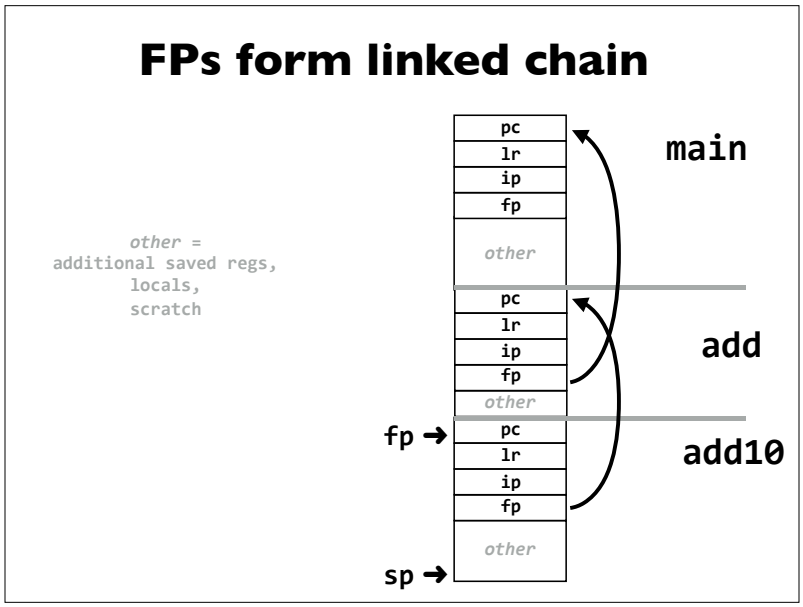
Body

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

Epilog

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





This figure shows how the frame pointers form a linked chain that connects the stack frames.

```
// start.s

// Need to initialize fp = NULL
// to terminate end of chain

    mov sp, #0x8000000
    mov fp, #0    // fp = NULL
    bl main
```

Recall that our start.s sets the stack pointer to the top of memory: 0x8000000. It also sets the frame pointer to 0, to show that there's no stack frame, this is the start of the stack. It's a "null pointer" for the stack.