Concepts So Far

ARM processor architecture

Assembly and machine language

Peripherals and GPIO

C language

- **Relationship between C and assembly**
- **Pointers and memory addresses**

Tools So Far

```
as
gcc
objcopy
objdump
hexdump

make
bash

[ld]
```

Where Are We Going?

Functions (today)

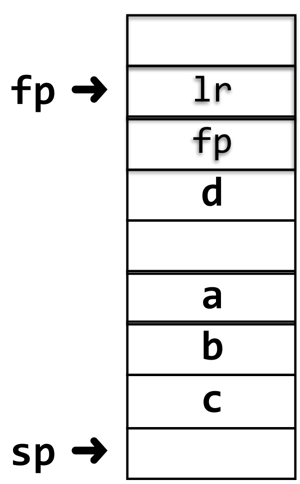
Serial communication and strings

Linking, loading, and starting

After this: the cool stuff from PL and DE

Calling Functions

Four important registers: `pc, lr, sp, fp`




```
int A(int a, int b, int c)
```

Topics

Calling functions:

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

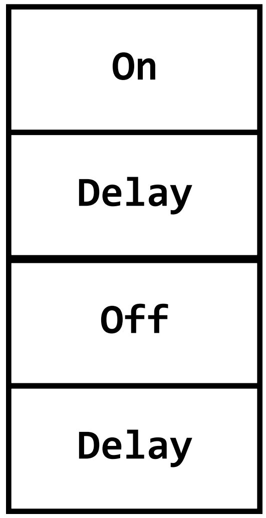
The stack and stack pointer (`sp`)

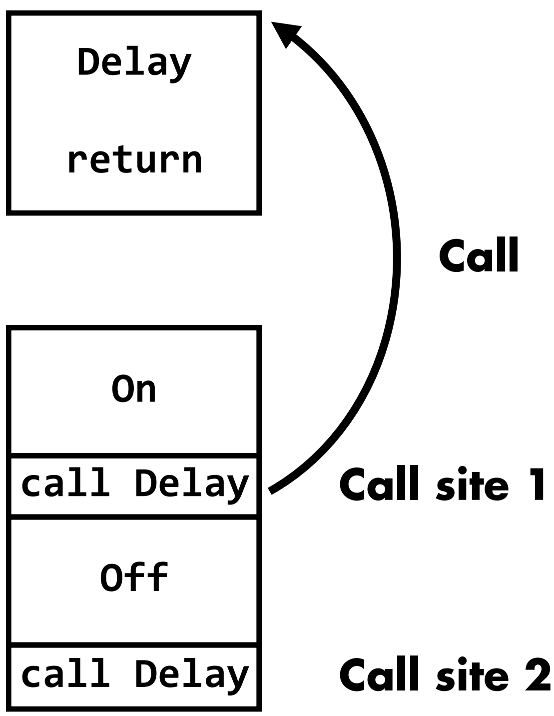
Activation records; the frame pointer (`fp`)

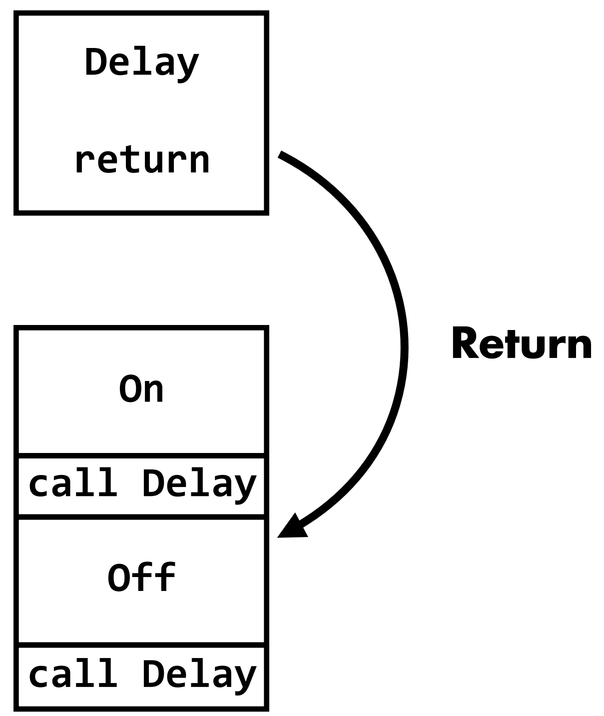
```
// Blink

// Setup GPIO 20
ldr r0, =0x20200008
mov r1, #1
str r1, [r0]
```

```
// turn on the led
mov r0, #(1<<20)
ldr r1, =0x2020001C // SET0
str r0, [r1]

// delay
mov r0, #0x3F0000
wait1:
    subs r0, #1
    bne wait1
```

```
// turn off the LED
mov r0, #(1<<20)
ldr r1, =0x20200028 // CLR0
str r0, [r1]

// delay
mov r0, #0x3F0000
wait2:
   sub r0, #1
   bne wait2

// delay would be a useful function
```


On
Delay
Off
Delay


Delay
return
Call
On
call Delay
Call site 1
Off
call Delay
Call site 2



Calling and Returning

```
// call

bl delay // branch and link : lr=pc+4


// return

bx lr // branch to register lr: pc=lr
```

```
// turn on the led
mov r0, #(1<<20)
ldr r1, =0x2020001C // SET0
str r0, [r1]

bl delay

// turn off the LED
mov r0, #(1<<20)
ldr r1, =0x20200028 // CLR0
str r0, [r1]

bl delay
```

```
// delay function
delay:
   mov r0, #0x3F0000
wait:
   sub r0, #1
   bne wait

   bx lr
```

```
// delay function
delay:
   mov r0, #0x3F0000
wait:
   sub r0, #1
   bne wait

   mov pc, lr
```

```
// turn on the led
mov r0, #(1<<20)
ldr r1, =0x2020001C // SET0
str r0, [r1]

mov r0, #0x3F0000
bl delay

// turn off the LED
mov r0, #(1<<20)
ldr r1, =0x20200028 // CLR0
str r0, [r1]

mov r0, #0x3F0000
bl delay
```

```
// delay function
delay:
   subs r0, #1
   bne delay

   bx lr
```

```
// turn on the led
mov r0, #20
bl set

mov r0, #0x3F0000
bl delay

// turn off the LED
mov r0, #20
bl clr

mov r0, #0x3F0000
bl delay
```

```
// set bit r0 in GPIO SET0 register
set:
    mov r1, #1
    lsl r0, r1, r0
    ldr r1, =0x2020001C // SET0
    str r0, [r1]
    bx lr

// clr bit r0 in GPIO CLR0 register
clr:
    mov r1, #1
    lsl r0, r1, r0
    ldr r1, =0x20200028 // CLR0
    str r0, [r1]
    bx lr
```

Argument Passing : ABI

r0-r3 are used for input arguments

r0-r1 are for return values

Application binary interface (ABI)

N.B. There are ways to pass >4 args

N.B. ARM uses extended ABI (eabi) as in arm-none-eabi

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


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

Register Conventions

Nomenclature

- **caller - calling function**
- **callee - function called**

r0-r3 callee-owned registers

- **Callee can change these registers**
- **Caller should not assume the values are the same when the function returns**

Register Conventions

r4-r15 caller-owned, callee-save registers

r11 (fp) - special

r12 (ip) - special (scratch register)

r13 (sp) - special

r14 (lr) - special

r15 (pc) - special

Discuss

The ARM architecture includes a *mechanism,* the instruction (`bl label`) to call a function. Propose a *convention* to call a function using other instructions.

What is the advantage of making most registers callee-saved?

Callee-Saved

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

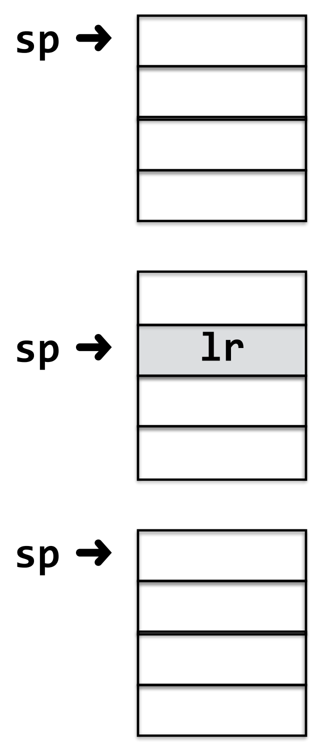
Caller doesn't need to make any assumptions of what the callee does

Where does the callee save registers?

```
// recursive delay function
recursive_delay:
   subs r0, #1
   blne recursive_delay
   bx lr

// Does this work?
```

The Stack

```
// init
mov sp, #0x8000
…

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

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



“Full Descending” Stack

```
// recursive delay function
recursive_delay:
  push {lr}           // *--sp=lr
  subs r0, #1
  blne recursive_delay
  pop {lr}            // lr=*++sp
  bx lr
```

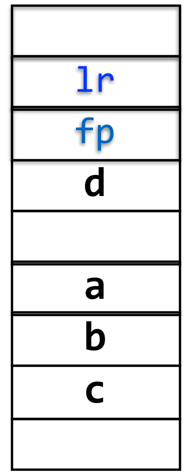
```
// delay function using return stack
recursivedelay:
  str lr, [sp, #-4]!  // *--sp=lr
  subs r0, #1
  blne recursivedelay
  ldr pc, [sp], #4    // pc=*sp++
```

Using the Stack w/ Functions

Saving and restoring registers

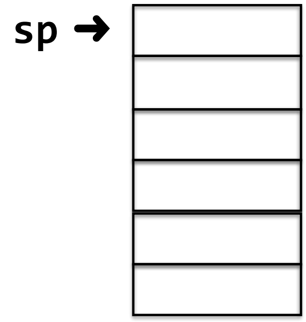
Local variables

*** Passing extra arguments (>4)**

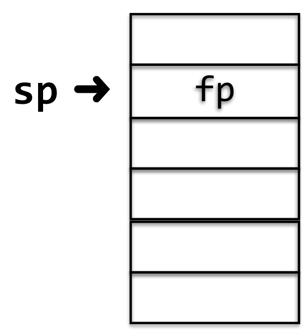



```
int A(int a, int b)
{
    return a - b;
}

int B(int a, int b, int c)
{
   int d = a + 2;
   return d + A(b, c);
}
```

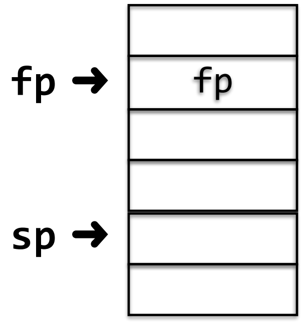
```
A:
push    {fp}
add fp, sp, #0
sub sp, sp, #12
str r0, [fp, #-8]
str r1, [fp, #-12]
ldr r2, [fp, #-8]
ldr r3, [fp, #-12]
rsb r3, r3, r2
mov r0, r3
sub sp, fp, #0
pop {fp}
bx  lr
```



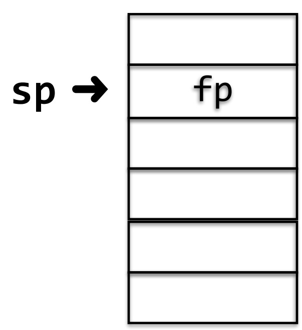

```
A:
push    {fp}
add fp, sp, #0
sub sp, sp, #12
str r0, [fp, #-8]
str r1, [fp, #-12]
ldr r2, [fp, #-8]
ldr r3, [fp, #-12]
rsb r3, r3, r2
mov r0, r3
sub sp, fp, #0
pop {fp}
bx  lr
```



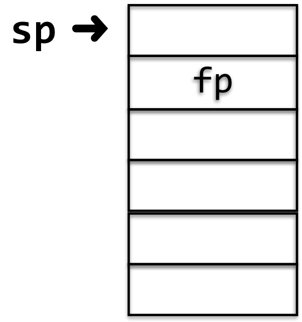
```
A:
push    {fp}
add fp, sp, #0
sub sp, sp, #12
str r0, [fp, #-8]
str r1, [fp, #-12]
ldr r2, [fp, #-8]
ldr r3, [fp, #-12]
rsb r3, r3, r2
mov r0, r3
sub sp, fp, #0
pop {fp}
bx  lr
```



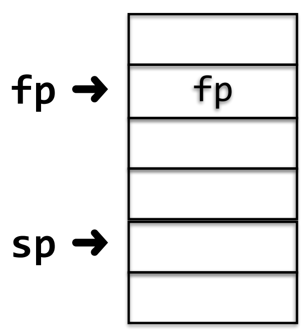

```
A:
push    {fp}
add fp, sp, #0
sub sp, sp, #12
str r0, [fp, #-8]
str r1, [fp, #-12]
ldr r2, [fp, #-8]
ldr r3, [fp, #-12]
rsb r3, r3, r2
mov r0, r3
sub sp, fp, #0
pop {fp}
bx  lr
```



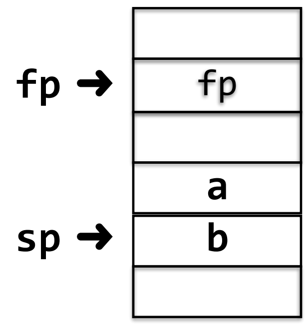
```
A:
push    {fp}
add fp, sp, #0
sub sp, sp, #12
str r0, [fp, #-8]
str r1, [fp, #-12]
ldr r2, [fp, #-8]
ldr r3, [fp, #-12]
rsb r3, r3, r2
mov r0, r3
sub sp, fp, #0
pop {fp}
bx  lr
```




```
A:
push    {fp}
add fp, sp, #0
sub sp, sp, #12
str r0, [fp, #-8]
str r1, [fp, #-12]
ldr r2, [fp, #-8]
ldr r3, [fp, #-12]
rsb r3, r3, r2
mov r0, r3
sub sp, fp, #0
pop {fp}
bx  lr
```

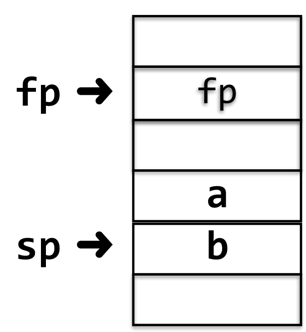


```
A:
push    {fp}
add fp, sp, #0
sub sp, sp, #12
str r0, [fp, #-8]
str r1, [fp, #-12]
ldr r2, [fp, #-8]
ldr r3, [fp, #-12]
rsb r3, r3, r2
mov r0, r3
sub sp, fp, #0
pop {fp}
bx  lr
```

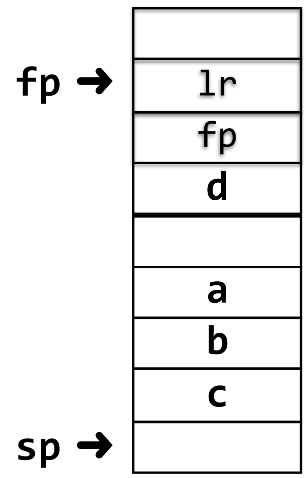


N.B. Extra space is allocated so that the sp is always aligned to 8 bytes

```
A:
push    {fp}
add fp, sp, #0
sub sp, sp, #12
str r0, [fp, #-8]
str r1, [fp, #-12]
ldr r2, [fp, #-8]
ldr r3, [fp, #-12]
rsb r3, r3, r2
mov r0, r3              // return a-b
sub sp, fp, #0
pop {fp}
bx  lr
```

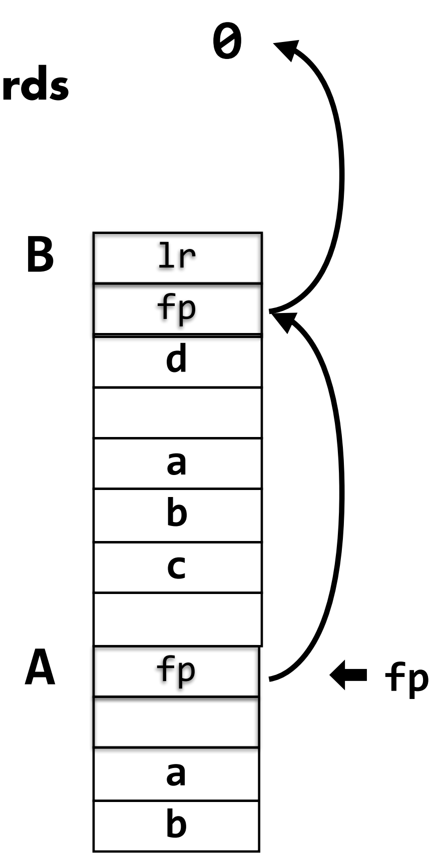


```
B:
push    {fp, lr}
add fp, sp, #4
sub sp, sp, #24
str r0, [fp, #-16]
str r1, [fp, #-20]
str r2, [fp, #-24]
ldr r2, [fp, #-16]
add r3, r3, #2
str r3, [fp, #-8]
ldr r0, [fp, #-20]
ldr r1, [fp, #-24]
bl  A
mov r2, r0
ldr r3, [fp, #-8]
add r3, r2, r3
mov r0, r3
sub sp, fp, #4
pop {fp, pc}
```



Activation Records

Frame stores
lr
fp



```
int A(int a, int b) { return a-b; }

% arm-none-eabi-gcc -O2

A:
    rsb r0, r1, r0
    bx lr
```

Why fp?

1. Print "backtrace" for debugging
   - Program crashes; what happened?
2. Nested function scopes
   - Define function inside function
   - Inside function can refer to outer function's variables
3. Unwinding the stack
   - Exceptions, ...

`ldm` and `stm`

```
push {r0, r1}

// f = full, d = descending
stmfd sp!, {r0, r1}
// d = decrement, b = before
stmdb sp!, {r0, r1}

str r1, [sp, #-4]!
str r0, [sp, #-4]!

// stm works with up to 8 registers
```

```
pop {r0, r1}

// f = full, d = descending
ldmfd sp!, {r0, r1}
// i = increment, a = after
ldmia sp!, {r0, r1}

ldr r0, [sp], #4
ldr r1, [sp], #4

// ldm works with up to 8 registers
```