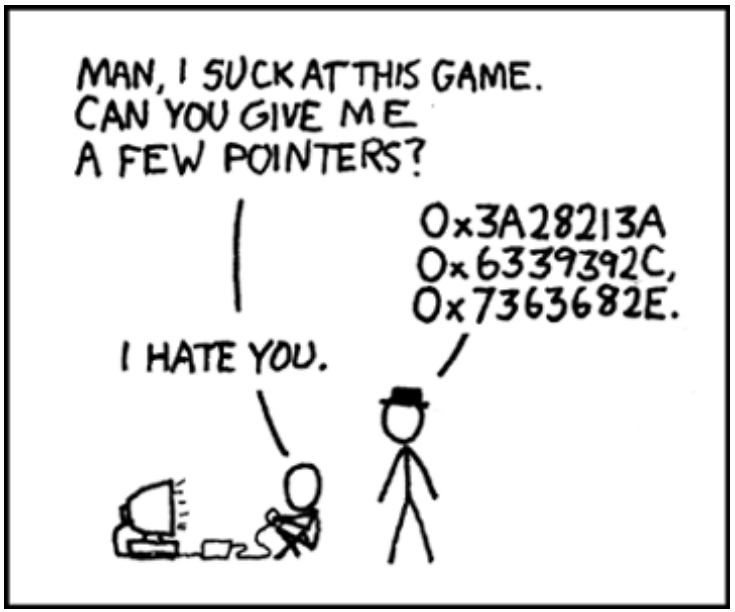
Goals for today

- Review pointer operations (from last time)
- ARM addressing modes, translation to/from C
- Implementation of C function calls
- Management of runtime stack, register use



Pointer follow up

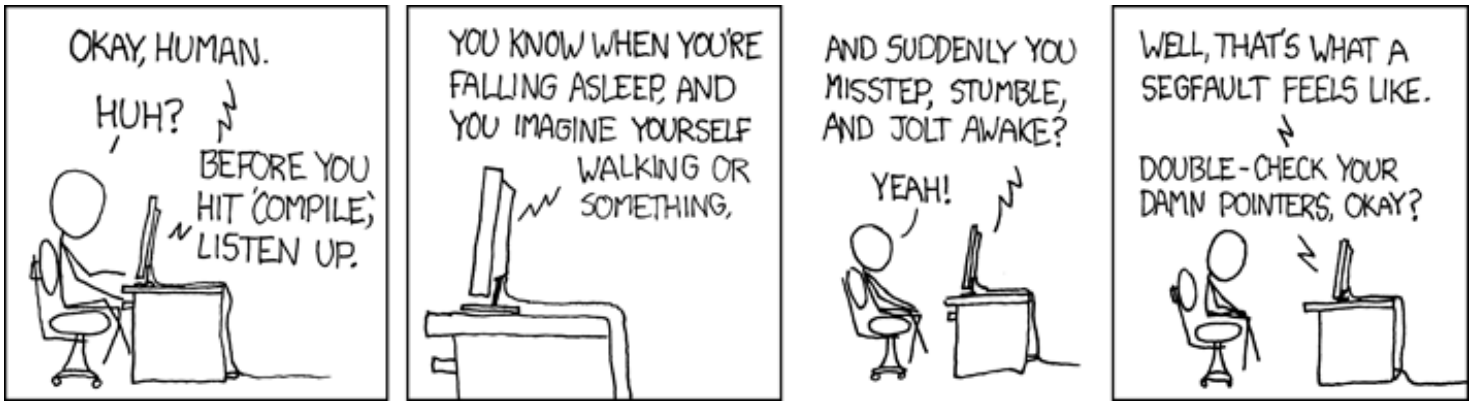
```
int arr[4], *p;
p = arr;

// Are these lines same or not?

arr[1] = 107;
p[1] = 107;
*(p + 1) = 107;
*((char*)p + 1) = 107;
*(char*)(p + 1) = 107;
```

Pointers: the fault in our *s

Pointers are ubiquitous in C, and inherently dangerous. Watch out!

Q. For what reasons might a pointer be invalid?

Q. What is consequence of using an invalid pointer?



Be wary of optimization

```
void wait_til_zero(int *ptr)
{
   while (*ptr != 0)
      ;       // loads *ptr each iteration?
}
```

Optimizer can be clever & surprisingly aggressive. Look at asm to confirm what it did!

Can block it by using function as black-box, qualify as volatile, or add memory barrier

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
  ...
```

How do these modes map to C language features?

```
                // excerpted from blink.s
loop:
  ldr r0, =0x2020001C    // set pin high
  str r1, [r0]
  mov r2, #0x3F0000      // delay
  subs r2, #1
  bne .-4
  ldr r0, =0x20200028    // set pin low
  str r1, [r0]
  mov r2, #0x3F0000      // delay
  subs r2, #1
  bne .-4
  b loop
```

```
    ldr r0, =0x2020001C
    str r1, [r0]
    b delay
    ldr r0, =0x20200028
    str r1, [r0]
    b delay
    b loop

delay:
    mov r2, #0x3F0000
    subs r2, #1
    bne .-4
  // but...
      how to return when loop finished?
```

```
    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
    subs r2, #1
    bne .-4
    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 >> 1
    mov r14, pc
    b delay
    b loop

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

We've just invented our own parameter passing!

Anatomy of C function call

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

int rec_fact(int n)
{
   if (n <= 1) return 1;
   return n*rec_fact(n-1);
}
```

Call and return (possibly nested)

Pass parameters (by value)

Local variables

Return value

Scratch/working space

Able to operate cross-module

Application binary interface

An ABI specifies requirements for code to interoperate:

- **Mechanism for call/return**
- **How parameters passed**
- **How return value communicated**
- **Use of registers**
- **Stack management**
- **We are using ARM eabi (embedded abi)**

Mechanics of call/return

Caller puts arguments in r0-r3
Call instruction is `bl` (branch and link)

```
mov r0, #100
mov r1, #7
bl sum          // lr=pc-4
```

Callee puts return value in r0
Return instruction is `bx` (branch exchange)

```
add r0, r0, r1
bx lr           // pc=lr
```

Register use

- `r0-r3` **general-purpose, callee-owned, caller-saved**
  callee can change value
  caller should not expect to be same after call returns

- `r4-r11` **general-purpose, caller-owned, callee-saved**
  callee must save/restore existing value if changing
  caller can expect to be same after call returns

- `r12-r15` **special-purpose, mostly treated callee-saved**

- **stack used as scratch space for save/restore registers**

Need for a stack

What if called function wants to make call of its own? (i.e. callee becomes caller)

Only one link register — what to do?

What about other register contention?

What if more than 4 parameters?

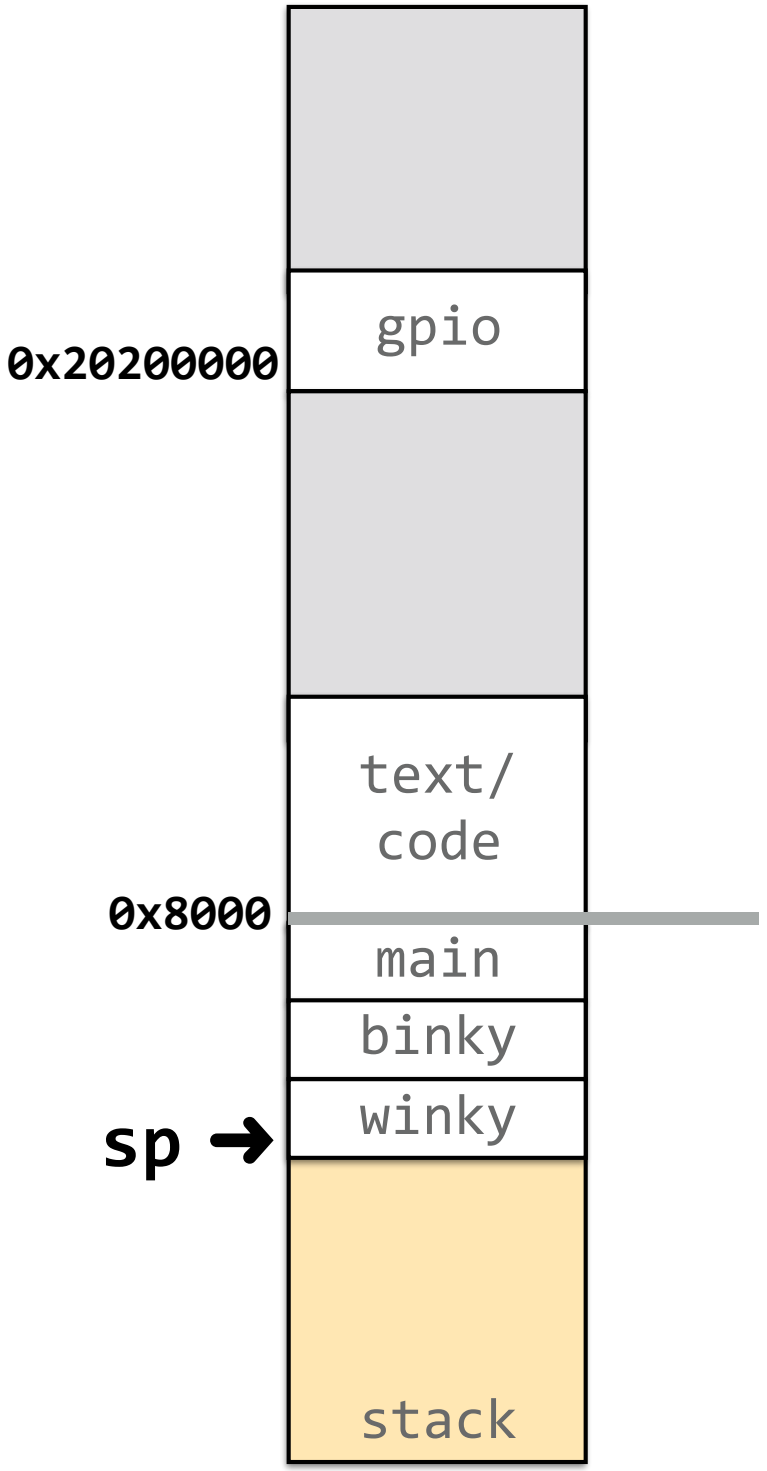
And where do locals go?…

The stack

Region in memory to stash values, save regs, scratch space, local variables

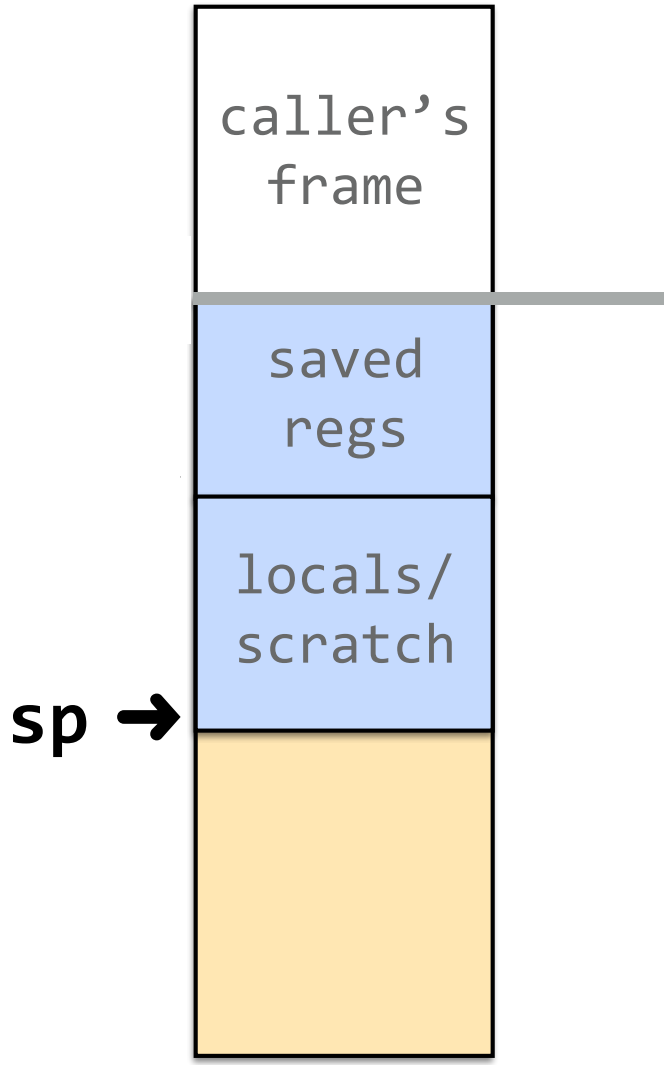
- **LIFO: push adds value on top of stack, pop removes lastmost value**
- `r13` **(alias** `sp`**) points to topmost value**
- **stack grows down**
  - **newer values at lower addresses**
  - **push subtracts from** `sp`
  - **pop adds to** `sp`
- **push/pop are aliases for a general instruction (load/store multiple with writeback)**

```
void main(void)
{
   binky(3);
}

void binky(int a)
{
   winky(10, a);
}

void winky(int x, int y)
{
  ...
```



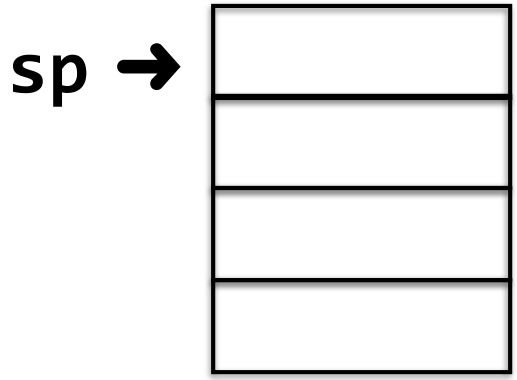
Single stack frame

```
int winky(int x, int y)
{
   int z;
   ...
   return 3;
}
```

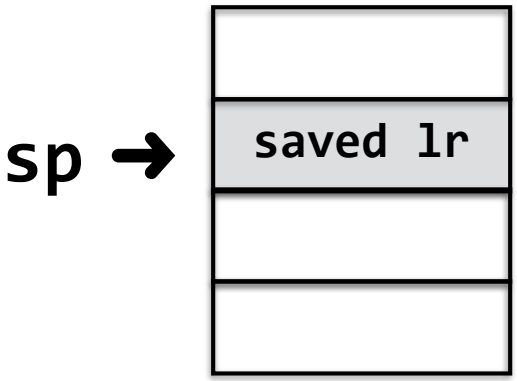


Stack operations

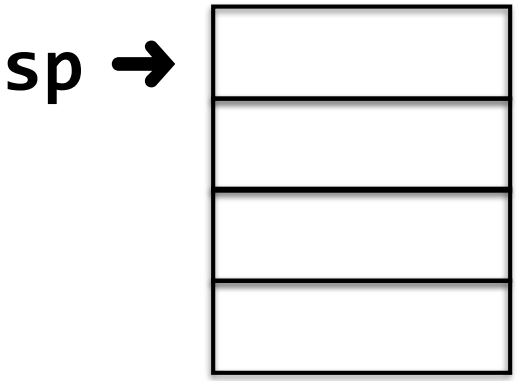
```
// init
mov sp, #0x8000
```




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




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



"Full Descending" Stack

```
recursive_delay:
  push {lr}
  subs r0, #1
  blne recursive_delay
  pop {lr}
  bx lr
```

At function entry, push used to save registers values that must be preserved

At exit, pop used to restore

push/pop allows any subset of registers

What is pushed/pop by these functions?

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

int winky(int x) {
    return binky(x, 87);
}

int dinky() {
    return winky(5) + winky(22);
}

int pinky(int x) {
    return winky(x) + winky(7) + x;
}
```

Are locals/params stored on stack? sometimes...

```
int megaparam(int a, int b, int c,
              int d, int e, int f);

int megalocal(void) {
    int arr[100];
    ...

void use_address(int a) {
    fn(&a);
```

Maintaining a frame pointer

Provides “anchor” into frame

Enables:

- backtrace for debugging
- unwind stack on exception
- nested function scope

Mechanics of a frame pointer

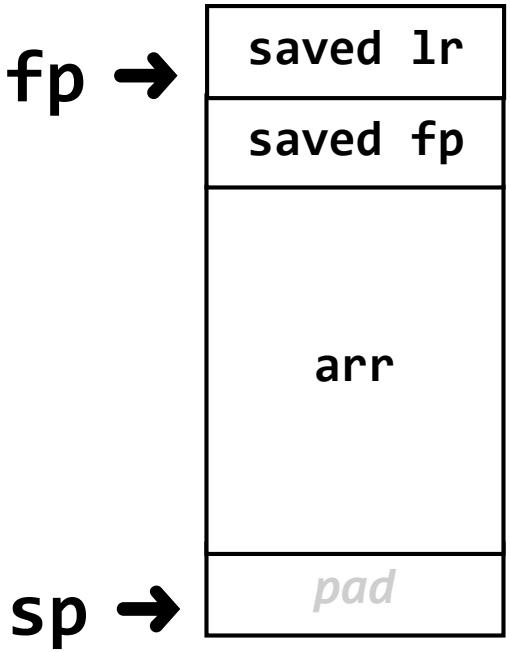
gcc flag `-fno-omit-frame-pointer`

`r11` used as `fp`

at function entry, set fp to first word used in current stack frame (at highest address)

must preserve previous fp (push/pop)

follow saved fp to trace backwards



fp is "anchor" into frame

Access stack contents fp-relative

```
int A(int x)
{
   int arr[5];

  return B(arr[4]);
}
```

```
A:
 push  {fp, lr}
 add   fp, sp, #4
 sub   sp, sp, #24
 ldr   r0, [fp, #-8]
 bl    B
 sub   sp, fp, #4
 pop   {fp, lr}
 bx    lr
```

Activation records (stack frames)

other =
additional saved regs,
locals,
scratch