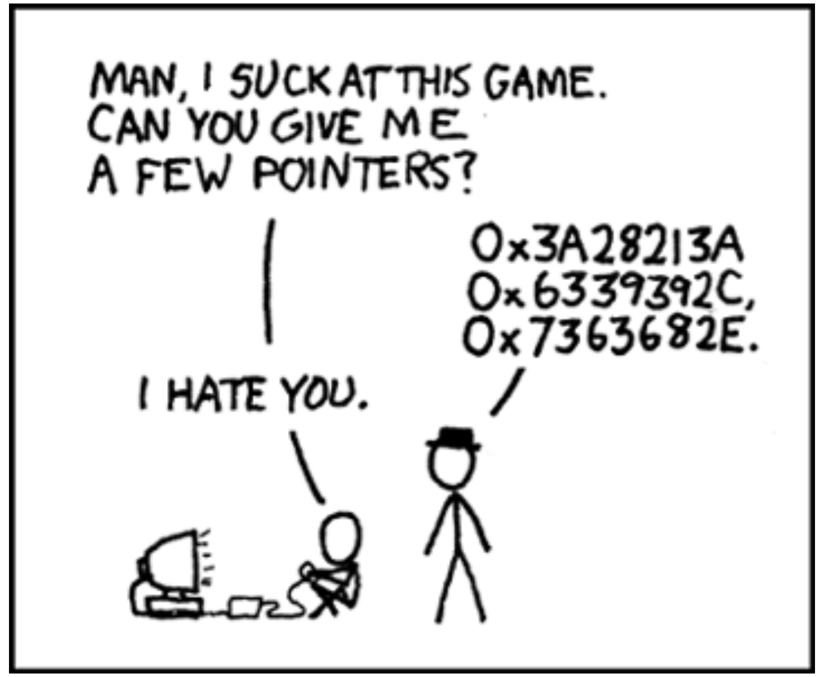
This week

- assign1 due Tues, have proved your bare-metal mettle!
- lab prep
  - read info on 7-segment display
  - bring your tools if you have them



Goals for today

- Leftovers: volatile qualifier, bare-metal build
- ARM addressing modes, translation to/from C
- Pointers, pointers, and more pointers!

When coding directly in assembly, the instructions you see are the instructions you get, no surprises!

For C source, you may need to drop down to see what compiler has generated to be sure of what you're getting

What transformations are *legal* ?
What transformations are *desirable* ?

```
int i, j;

i = 1;
i = 2;
j = i;

// can be optimized to

i = 2;
j = i;

// is this ever not equivalent/ok?
```

button.c

The little button that wouldn’t

`volatile`

For an ordinary variable, the compiler has complete knowledge of when it is read/written and can optimize those accesses as long as it maintains correct behavior.

However, for a variable that can be read/written externally (by another process, by peripheral), these optimizations will not be valid.

The **`volatile`** qualifier applied to a variable informs the compiler that it cannot remove, coalesce, cache, or reorder references. The generated assembly must faithfully execute each access to the variable as given in the C code.

Peripheral Registers

These registers are mapped into the address space of the processor (memory-mapped IO).

These registers may behave differently than memory.

For example: Writing a 1 into a bit in a SET register causes 1 to be output; writing a 0 into a bit in SET register does not affect the output value. Writing a 1 to the CLR register, sets the output to 0; write a 0 to a clear register has no effect. Neither SET or CLR can be read. To read the current value use the LEV (level) register.

Build process for bare-metal

The default build process for C assumes a *hosted* environment.

What does a hosted system have that we don't?

- standard libraries
- standard start-up sequence

To build bare-metal, our makefile disables these defaults; we must supply our own replacements where needed

Makefile settings

Compile freestanding

```
CFLAGS =-ffreestanding
```

Link without standard libs and start files

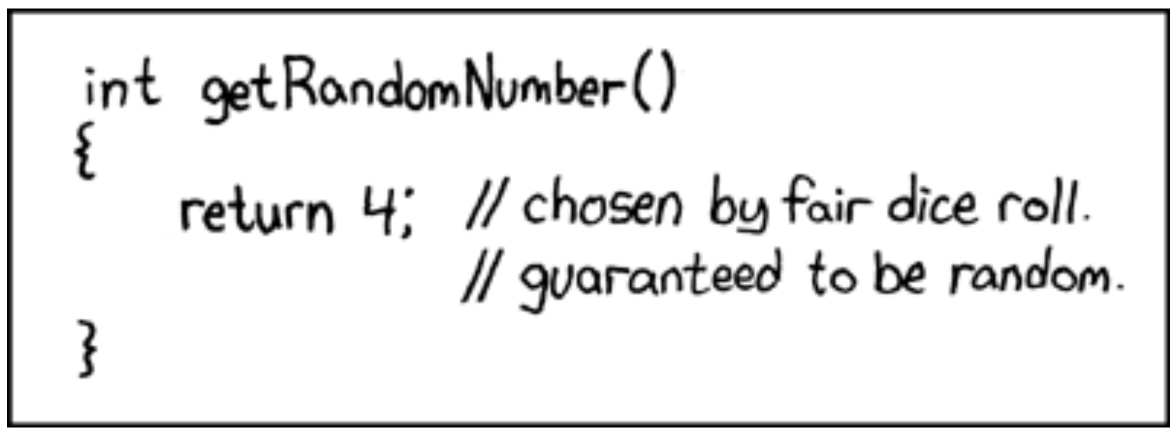
```
LDFLAGS = -nostdlib
```

Link with gcc to support division

```
LDLIBS = -lgcc
```

Must supply own replacement for libs/start

That’s where the fun is…!



LDR pseudo-instruction

Which of these **mov** instructions are valid?

```
mov r0, #0x7e
mov r0, #0x7e00000
mov r0, #0xfff00
mov r0, #0xffffffff
mov r0, #0x107e
```

For **ldr=** pseudo-instruction, any 32-bit constant is valid. If we replace **mov** with **ldr=,** what does assembler emit?

```
ldr r0, =0x107e
```

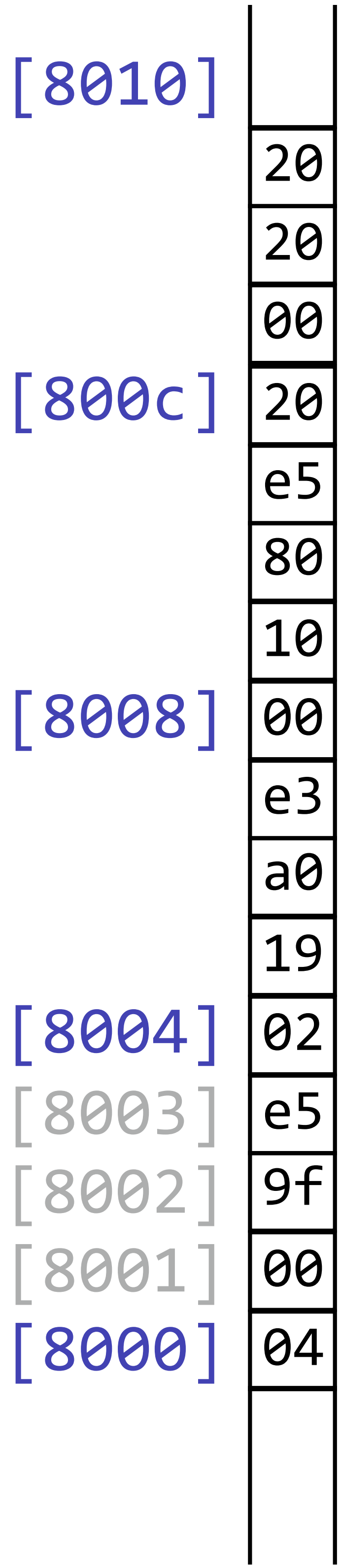
Program to set gpio 47 high (green led on Pi)

```
ldr r0, =0x20200020
mov r1, #(1<<15)
str r1, [r0]
```

Assembler emits:

```
e59f0004 ldr  r0, [pc, #4]
e3a01902 mov  r1, #32768  ; 0x8000
e5801000 str  r1, [r0]
20200020 .word  0x20200020
```

Memory as a linear sequence of indexed bytes



Same memory,
grouped into 4-byte words

[800c]	20200020
[8008]	e5801000
[8004]	e3a01902
[8000]	e59f0004

Note litte-endian byte ordering

ARM load/store instructions

```
ldr r0, [r1]

str r0, [r1]
```

Store is a misfit among ARM instructions — operands are in order of src, dst (reverse of all other instructions)

Fancier addressing modes

Preindex, non-updating

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

Preindex, writeback (update dst before use)

```
ldr r0, [r1, #4]!          // r1 pre-updated += 4
ldr r0, [r1, r2]!          // r1 pre-updated += r2
ldr r0, [r1, r2, lsl #4]!  // r1 pre-updated += r2 << 4
```

Postindex (update dst after use)

```
ldr r0, [r1], #4           // r1 post-updated += 4
ldr r0, [r1], r2           // r1 post-updated += r2
ldr r0, [r1], r2, lsl #4   // r1 post-updated += r2 << 4
```

C Pointer types

An ***address*** is a memory location, represented as an unsigned int. A ***pointer*** is a variable that holds an address. The "***pointee***" is the data stored at that address

At asm level, a 4-byte word could represent an address, an int, 4 characters, an ARM instruction,… The **ldr**/**str** instructions are agnostic to type. Asm has no type system to guide or restrict us on what we do with those words.

In C, there is a type system tracking the type of each variable. It further distinguishes pointer variables by the type of pointee. Operations are expected to respect the data type. Some examples:

- can't multiply `int*`'s, can't deference an `int`
- `int+int` is integer math, `int*+int` is pointer math, `double*+int` is pointer math (but not quite same as int*), `int*+int*` , `int*+double` and other such combos are illegal

Why Pointers?

Access specific memory locations like `FSEL2`

Pointers can be used to reference elements of an array

Pointers allow for creating dynamic data structures at runtime

Pointers can be used to efficiently share/pass references without making copies of large data structures

Pointers are used in data structures to reference other data structures

Pointer basics: & *

```
int m, n, *p, *q;

p = &n;
*p = n;          // same as prev line?

q = p;
*q = *p;         // same as prev line?

p = &m, q = &n;
*p = *q;
m = n;           // same as prev line?
```

Pointer and arrays

```
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

```
char *s = "Stanford";
char arr[] = "University";
char oldschool[] = {'L','e','l','a','n','d'};
char buf[100];
char *ptr;


 ptr = s;      // which assignments are valid?
 ptr = arr;
 ptr = buf;
 arr = ptr;
 buf = oldschool;
```

\0
64
63
61
6c
65
4c

??\06463
616c654c

What does a typecast <u>actually do</u>?

```
int *p; double *q; char *s;

ch  = *(char *)p;
val = *(int *)s;
val = *(int *)q;
```

Aside: why is this even allowed?

Casting between different types of pointers — perhaps plausible

Casting between pointers and int — sketchy

Casting between pointers and float — bizarre

What does this program do

```
int main(int argc, char *argv[])
{
    unsigned int *start = 0x0,
                 *end = (unsigned int *)0x100000;

    for (unsigned int *addr = start; addr < end; addr++
        *addr = 0;
}
```

… on linux?

… on MacOS?

… on your pi?

Pointers: the fault in our *s

Pointers are ubiquitous in C, and inherently dangerous. Be vigilant!

Q. For what reasons might a pointer be invalid?

Q. What is consequence of using an invalid pointer?