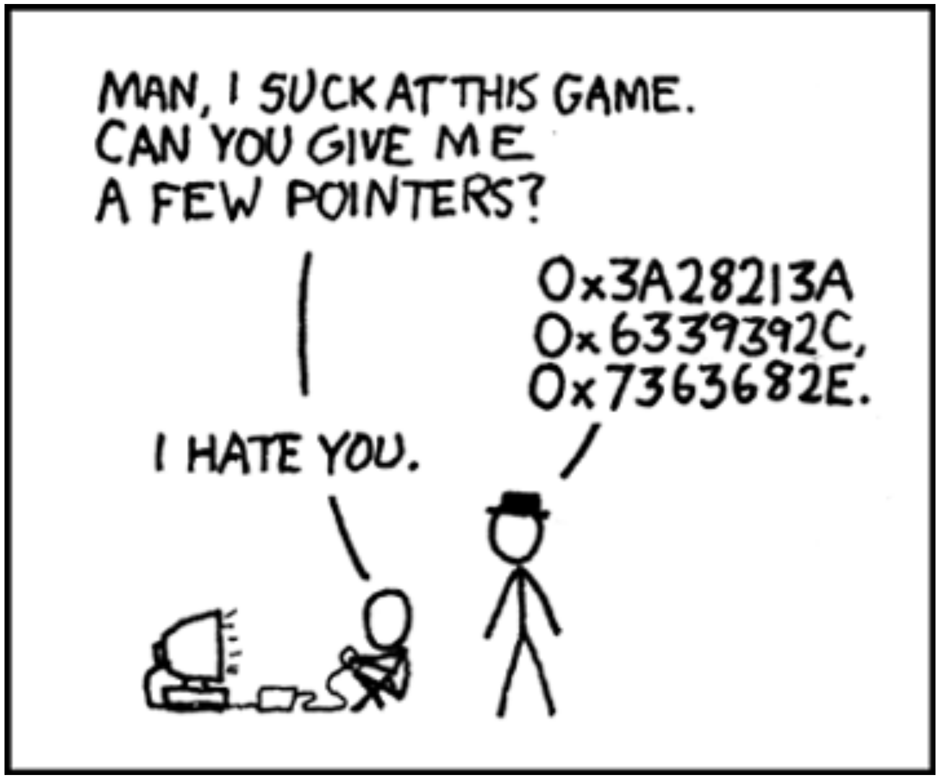
This week

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

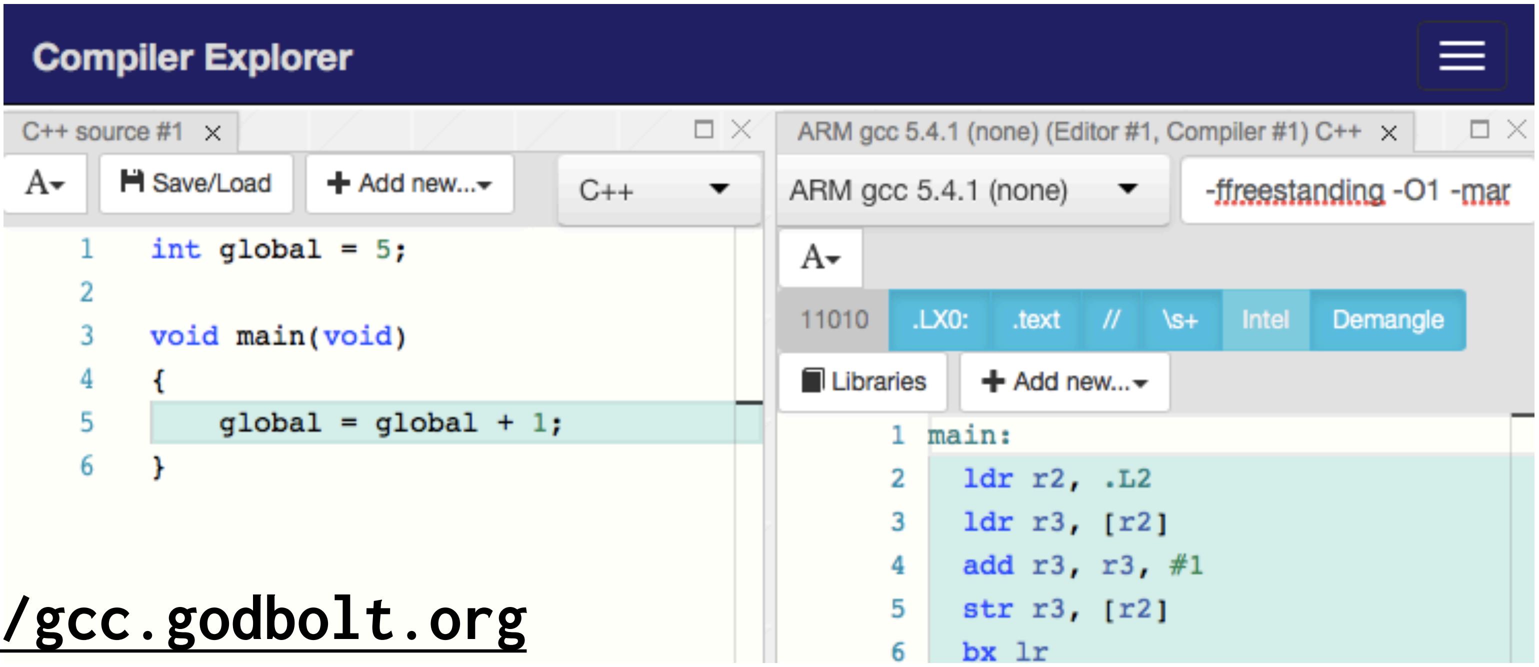

MAN, I SUCK AT THIS GAME. CAN YOU GIVE ME A FEW POINTERS?
0x3A28213A 0x6339392C, 0x7363682E.
I HATE YOU.

Goals for today

- Relate C <-> asm
- Bare-metal build
- ARM addressing modes
- Pointers, pointers, and more pointers!

C language features closely model the ISA: data types, arithmetic/logical operators, control flow, access to memory, ...

Compiler Explorer is a neat interactive tool to see translation



```
int global = 5;

void main(void)
{
    global = global + 1;
}
```

```
main:
  ldr r2, .L2
  ldr r3, [r2]
  add r3, r3, #1
  str r3, [r2]
  bx lr
```

https://gcc.godbolt.org

```
.equ DELAY, 0x3F0000

ldr r0, FSEL2
mov r1, #1
str r1, [r0]
mov r1, #(1<<20)

loop:
  ldr r0, SET0
  str r1, [r0]
  mov r2, #DELAY
  wait1:
      subs r2, #1
      bne wait1
  ldr r0, CLR0
  str r1, [r0]
  mov r2, #DELAY
  wait2:
      subs r2, #1
      bne wait2
  b loop

FSEL2: .word 0x20200008
SET0:  .word 0x2020001C
CLR0:  .word 0x20200028
```

→ **C?**

let's do it!

Know your tools!

Assembler (as)

Transform assembly code (text)
into object code (binary machine instructions)
Mechanical translation, few surprises

Compiler (gcc)

Transform C code (text)
into object code
(likely staged C-> asm -> object)
Complex translation, high artistry

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*?

cbutton.c

The little button that wouldn’t

A cautionary tale

(syllabus on course web page has links to code shown in class!)

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.

`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.

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
- OS services

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

Makefile settings for bare-metal

Compile freestanding

```
CFLAGS =-ffreestanding
```

Link without standard libs and start files

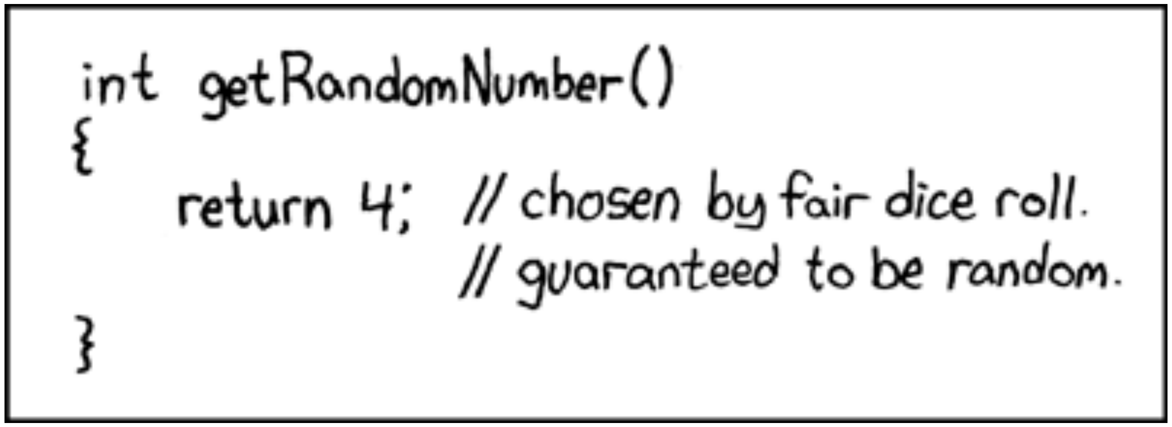
```
LDFLAGS = -nostdlib
```

Link with gcc to support division (no ARM divide instruction)

```
LDLIBS = -lgcc
```

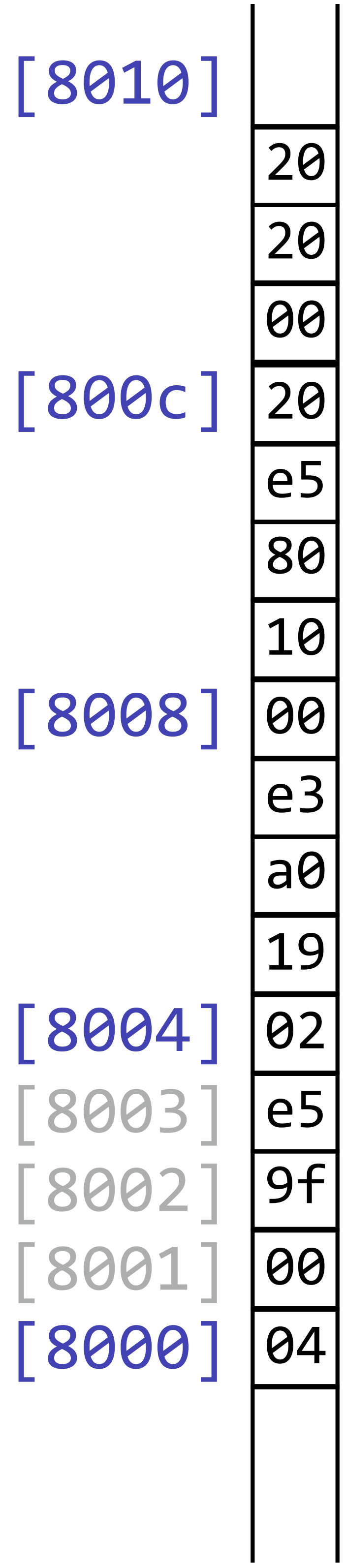
We must supply own replacement for libs/start

That's where the fun is…!

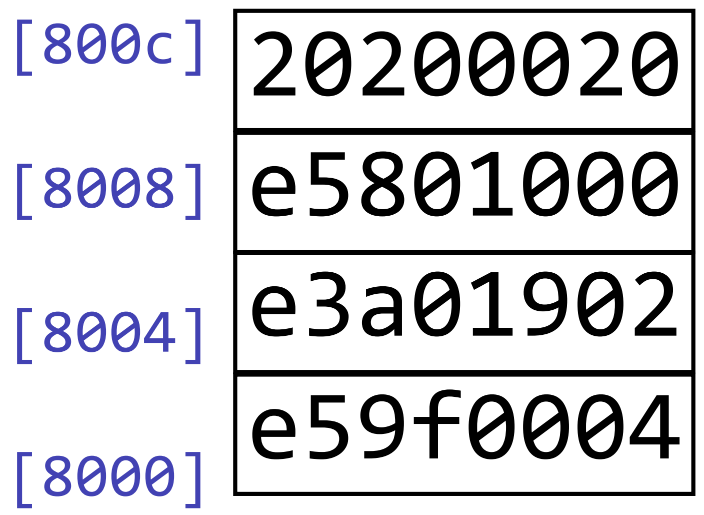


```
int getRandomNumber()
{
    return 4;  // chosen by fair dice roll.
               // guaranteed to be random.
}
```

Memory = linear sequence of indexed bytes



same bytes shown on left
but instead grouped in 4-byte words



Note litte-endian byte ordering

ARM load/store instructions

```
ldr r0, [r1]

str r0, [r1]
```

Load/store instructions are agnostic to type. They simply read or write a 4-byte word to/from the specified memory location. Did we just load an int, an address, 4 chars — who knows?

Fancy addressing modes for ldr/str

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
```

Why Pointers?

- Access specific memory by address, e.g. **FSEL2**
- Reference elements of an array/subarray
- Construct dynamic data structures at runtime
- Efficiently share/pass references without making copies of large data structures
- Coerce/manage type system when needed

CULTURE FACT:

IN CODE, IT'S NOT CONSIDERED RUDE TO POINT.

Pointer types in C

An ***address*** is a memory location, represented as an unsigned int

A ***pointer*** is a variable that holds an address

The "***pointee***" is the contents of memory at that address

C's type system tracks the type of each variable. It distinguishes pointer variables by the type of pointee. Operations are expected to respect the data type. In some cases we will need to typecast to coerce the compiler to go along with our plan.

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?
```

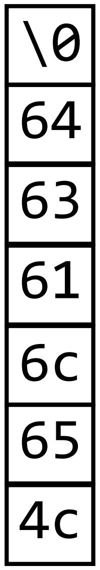
Pointer and arrays

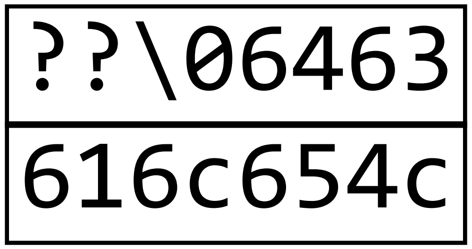
```
int n, arr[4], *p;

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

Nothing more than a pointer to a sequence of characters terminated by null char (zero byte)!

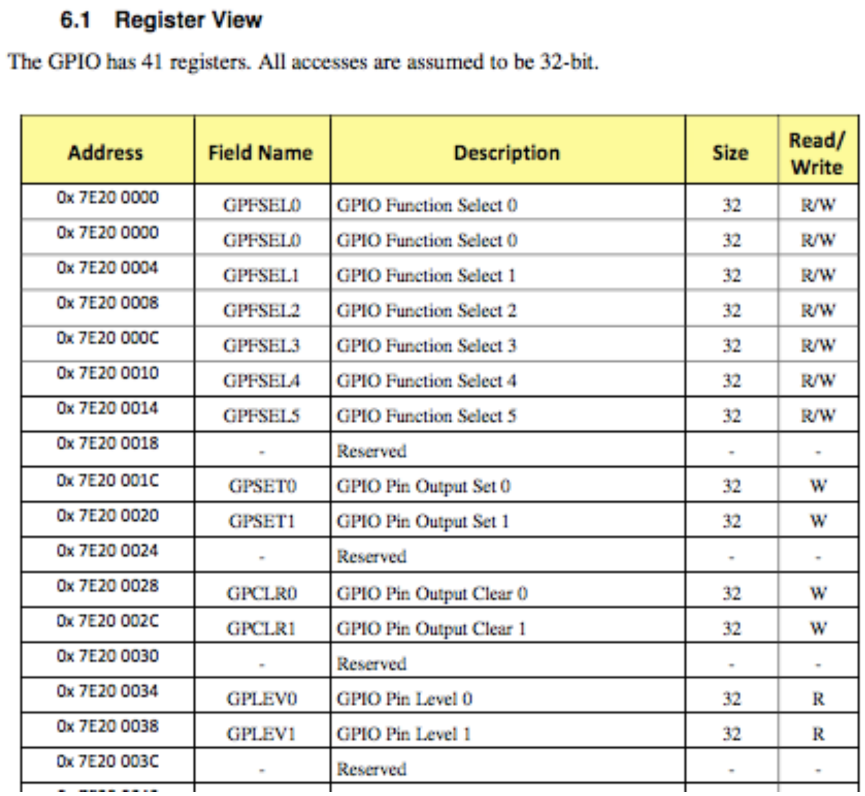
```
char *s = "Stanford";
char arr[] = "University";
char oldschool[] = {'L','e','l','a','n','d'};
char buf[100];
char *ptr;


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





Leveraging pointers & type system

```
struct gpio {
    unsigned int fsel[6];
    unsigned int reservedA;
    unsigned int set[2];
    unsigned int reservedB;
    unsigned int clr[2];
    unsigned int reservedC;
    unsigned int lev[2];
};
```



6.1 Register View

The GPIO has 41 registers. All accesses are assumed to be 32-bit.

Address	Field Name	Description	Size	Read/ Write
0x 7E20 0000	GPFSEL0	GPIO Function Select 0	32	R/W
0x 7E20 0000	GPFSEL0	GPIO Function Select 0	32	R/W
0x 7E20 0004	GPFSEL1	GPIO Function Select 1	32	R/W
0x 7E20 0008	GPFSEL2	GPIO Function Select 2	32	R/W
0x 7E20 000C	GPFSEL3	GPIO Function Select 3	32	R/W
0x 7E20 0010	GPFSEL4	GPIO Function Select 4	32	R/W
0x 7E20 0014	GPFSEL5	GPIO Function Select 5	32	R/W
0x 7E20 0018	-	Reserved	-	-
0x 7E20 001C	GPSET0	GPIO Pin Output Set 0	32	W
0x 7E20 0020	GPSET1	GPIO Pin Output Set 1	32	W
0x 7E20 0024	-	Reserved	-	-
0x 7E20 0028	GPCLR0	GPIO Pin Output Clear 0	32	W
0x 7E20 002C	GPCLR1	GPIO Pin Output Clear 1	32	W
0x 7E20 0030	-	Reserved	-	-
0x 7E20 0034	GPLEV0	GPIO Pin Level 0	32	R
0x 7E20 0038	GPLEV1	GPIO Pin Level 1	32	R
0x 7E20 003C	-	Reserved	-	-

```
volatile struct gpio *gpio = (struct gpio *)0x20200000;
gpio->fsel[0] = ...
```

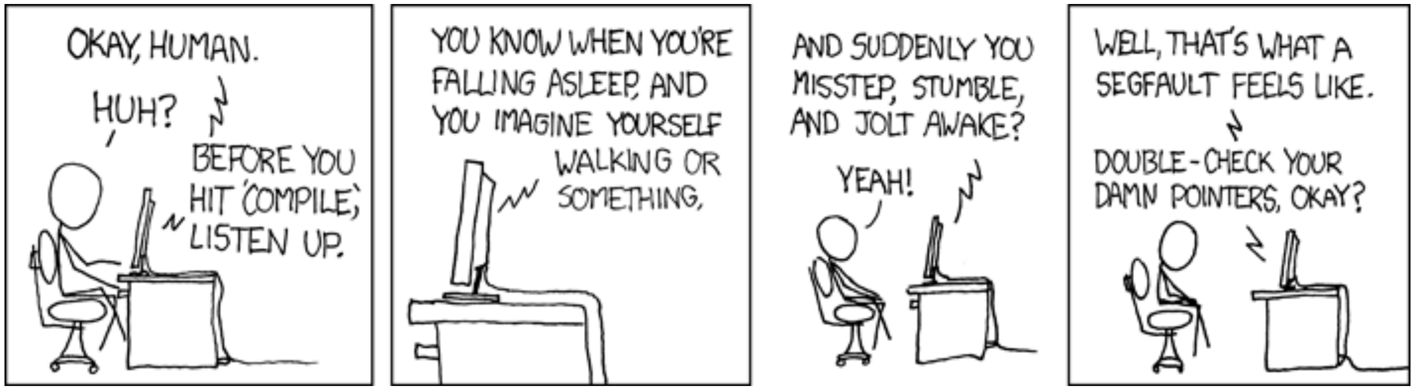
Advantages/disadvantages vs raw addresses?

Pointers: the fault in our *s

Pointers are ubiquitous in C and safety is lax. It is on you to be vigilant!

Q. For what reasons might a pointer be invalid?

Q. What are the consequences of using an invalid pointer?


OKAY, HUMAN.
HUH?
BEFORE YOU HIT 'COMPILE,' LISTEN UP.
YOU KNOW WHEN YOU'RE FALLING ASLEEP, AND YOU IMAGINE YOURSELF WALKING OR SOMETHING,
AND SUDDENLY YOU MISSTEP, STUMBLE, AND JOLT AWAKE?
YEAH!
WELL, THAT'S WHAT A SEGFAULT FEELS LIKE.
DOUBLE-CHECK YOUR DAMN POINTERS, OKAY?

"The fault, dear Brutus, is not in our stars,
But in ourselves, that we are underlings."
[Julius Caesar (I, ii, 140-141)]