This week

Assign1 due tomorrow

Congrats on having proved your bare-metal mettle!

Prelab for lab2

Read info on gcc/make and 7-segment display

Bring your tools if you have them!

"Gitting Started"

Ashwin Wed 4:30pm in B21

Hail the all-powerful C pointer!

Goals for today

Relate C <-> asm

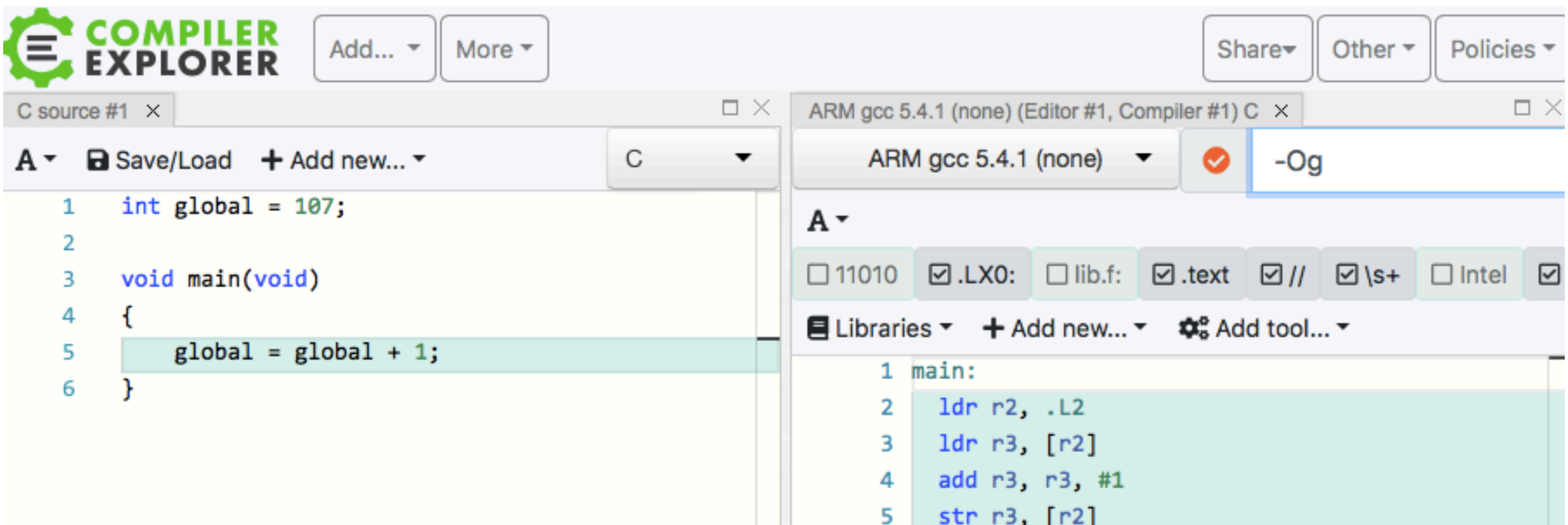
Bare-metal build

Pointers, pointers, and more pointers!

ARM addressing modes

Matt Godbolt's Compiler Explorer

Neat interactive tool to see translation from C to assembly.
Let's try it now!



https://gcc.godbolt.org

Configure settings to `ARM gcc 5.4.1(none), -Og` to follow along

```
.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?

blink.s → cblink.c

let's do it!

(Source code available in courseware repo at lectures/C1/code)

Know your tools

Assembler as

Transform assembly code (text)
into object code (binary machine instructions)

Mechanical rewrite, few surprises

Compiler gcc

Transform C code (text)
into object code
(likely staged C ➜ asm ➜ object)

Complex translation, high artistry

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

When coding in C, compiler transforms C source into assembly. Sometimes have to drop down to see what was generated to be sure of what you're getting

What transformations are legal ?
What transformations are desirable ?

`cblink.c`

The little LED that wouldn’t

A cautionary tale

(Source code available in courseware repo at lectures/C1/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 has no effect. Writing a 1 to the CLR register, sets the output to 0; writing a 0 to the CLR register has no effect. Neither SET or CLR can be read. To read the current value use the LEV (level) register.

Q: What can happen if the C compiler makes assumptions reasonable for ordinary memory that don't hold for these oddball registers?

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

Build settings for bare-metal

Compile freestanding

```
CFLAGS = -ffreestanding
```

Link without standard libs or start files

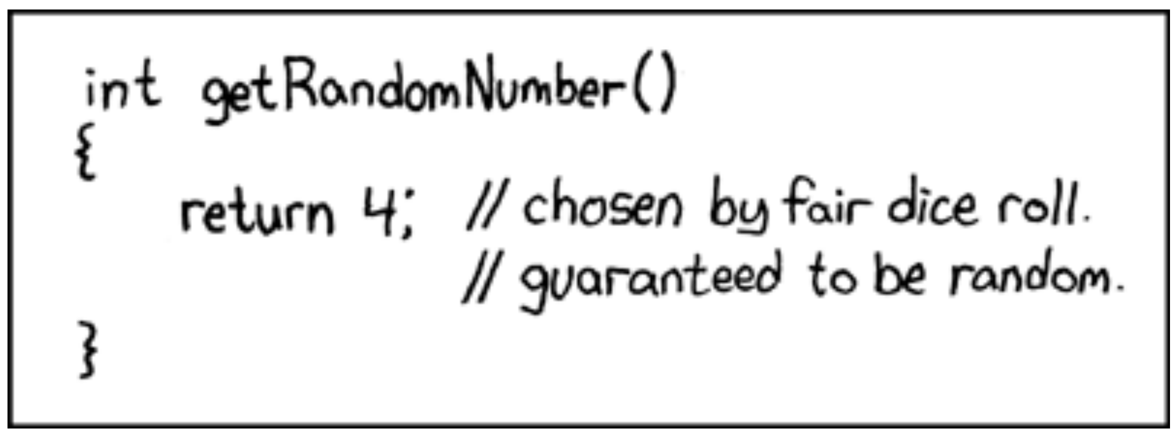
```
LDFLAGS = -nostdlib
```

Link with gcc if need division (b/c no ARM divide instruction)

```
LDLIBS = -lgcc
```

Write our own code for all libs and start files

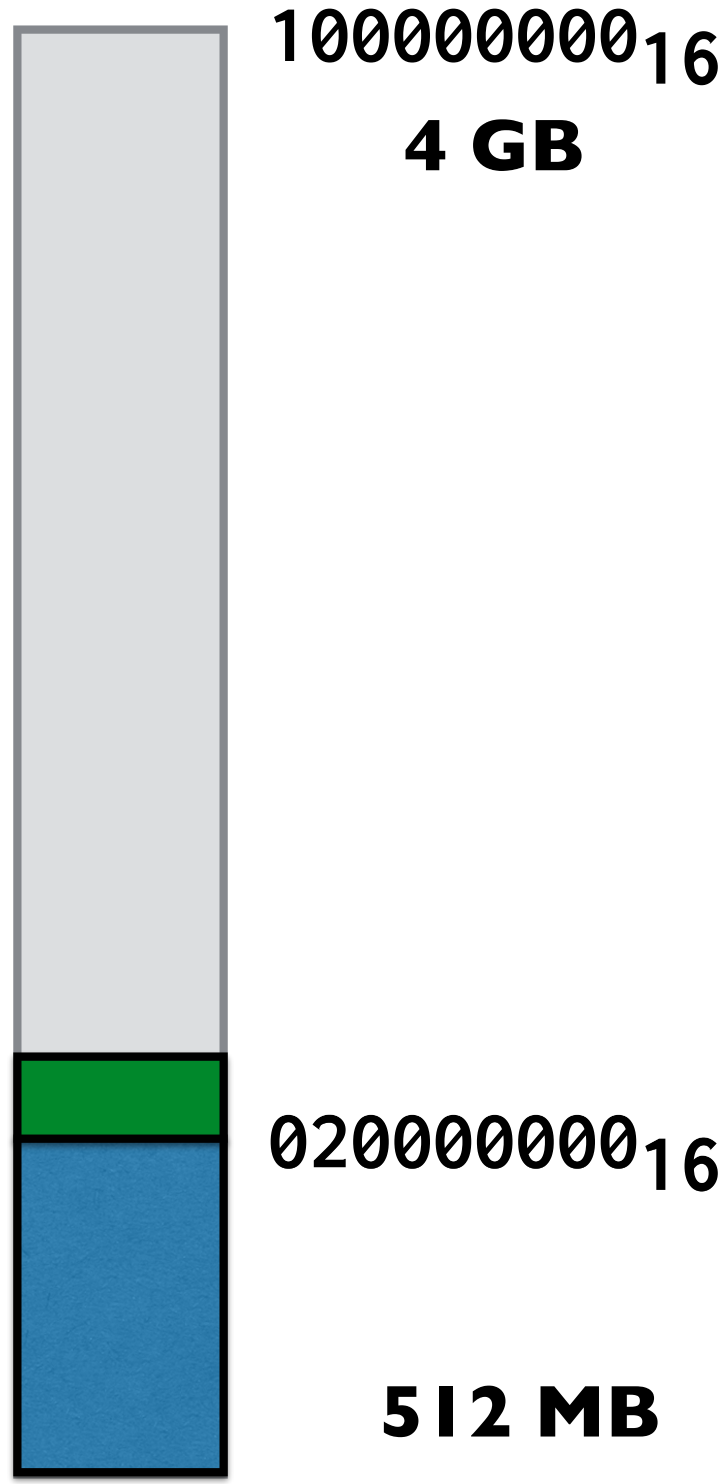
This puts us in an exclusive club…



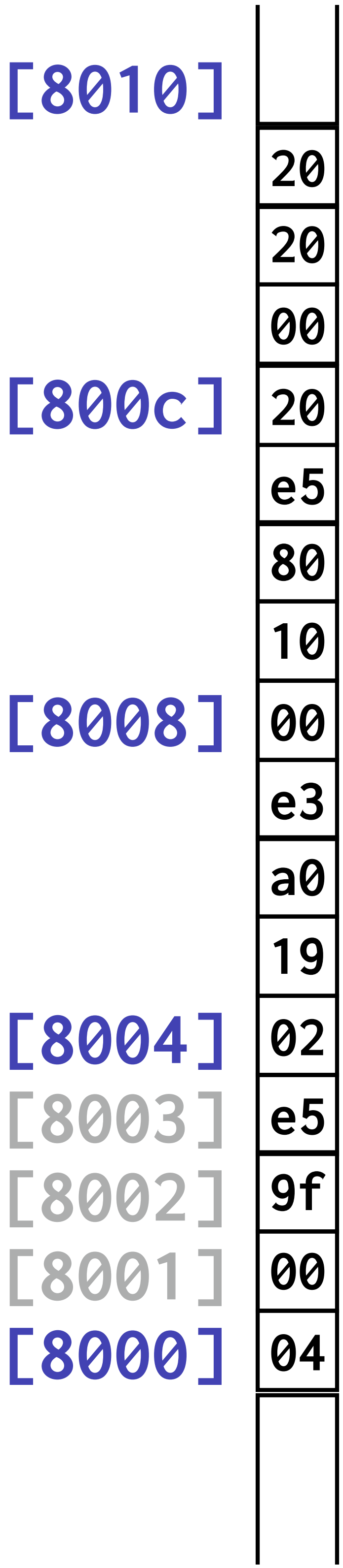
Memory

Memory is a linear sequence of bytes, indexed by address

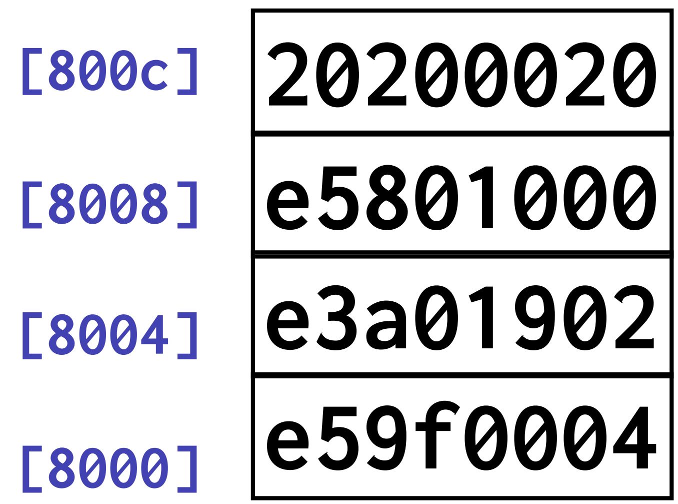
Addresses start at 0, go to 2^{31} -1 (32-bit architecture)



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]
```

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

ASM and memory

When loading a 4-byte word from memory, those bytes could represent:

- an address,
- an int,
- 4 characters
- an ARM instruction

There is no indication of the data type.
In fact, the `ldr` and `str` instructions are completely agnostic to type: assembly has no type system to guide or restrict what we do with the data being read/written.

Why pointers?

- Access specific memory by address, e.g. FSEL2
- Access data by its offset relative to other nearby data (array elements, struct fields)
- 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 vocabulary

An <u>address</u> is a memory location, represented as an unsigned int (because this is a 32-bit architecture).

A <u>pointer</u> is a variable that holds an address.

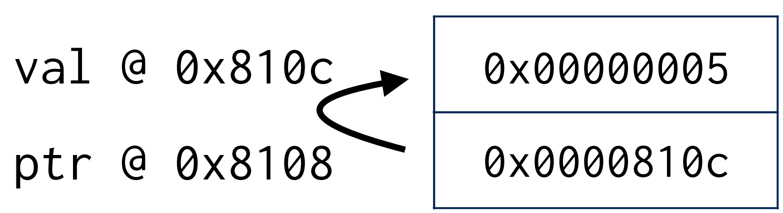
The "<u>pointee</u>" is the data stored at that address.

* is the <u>dereference</u> operator, & is <u>address-of</u>.

C code

```
int val = 5;
int *ptr = &val;
```

address **memory**



C pointer types

C has a *type system:* each variable has a declared type

Operations required to respect the data type

- Can't multiply int*'s, can't deference an int

Distinguishes pointer variables by type of pointee

- Dereferencing an int* is an int
- Dereferencing a char* is a char

Typecast can coerce different behavior from compiler

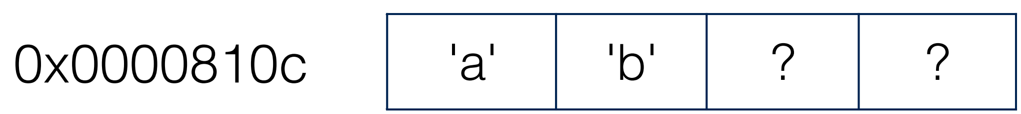
Pointer operations: & *

```
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?
```

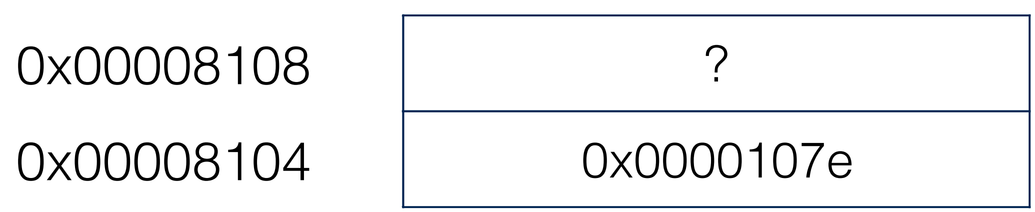
C arrays

An array is a contiguous sequence of homogeneous elements. An array declaration specifies element type and count of elements.

```c
char letters[4];
letters[0] = 'a';
letters[1] = 'b';
```




```c
int nums[2];
nums[0] = 0x107e;
```



Arrays and pointers

You can assign an array to a pointer:

```
int nums[2] = {5, 7};
int *ptr = nums;   // ptr = &(nums[0]);
```

An array and a pointer to the first element have a lot in common.

Pointer arithmetic operates in element-sized chunks, i.e., scales by `sizeof(type)`

```
ptr = ptr + 1; // now points to nums[1]
```

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

No real string data type, just a pointer to a sequence of characters terminated by null char (zero byte)!

	0x63	0x64	0x00	??
oldschool	0x4c	0x65	0x6c	0x61

```
char *s = "Stanford";
char arr[] = "University";
char oldschool[] = {'L','e','l','a','n','d','\0'};
char buf[20];
char *ptr;


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

Fancy ARM 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)

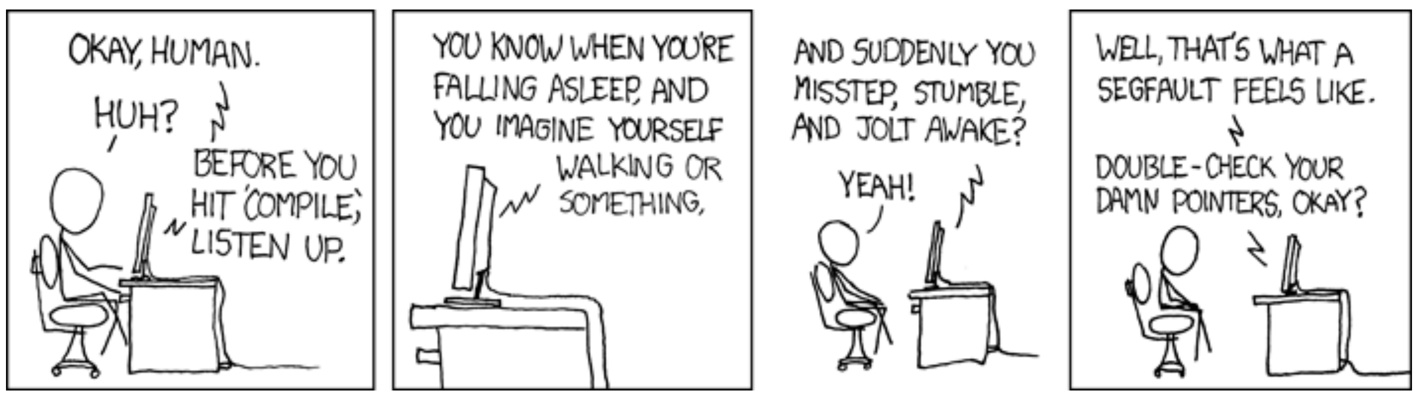
```
ldr r0, [r1], #4          // r1 post-updated += 4
ldr r0, [r1], r2          // r1 post-updated += r2
ldr r0, [r1], r2, lsl #4  // r1 post-updated += r2 << 4
```

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?



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