This week

- Assignment 1 due Wednesday: you'll have proved your bare-metal mettle!
- Lab 2:
  - do pre-lab reading!
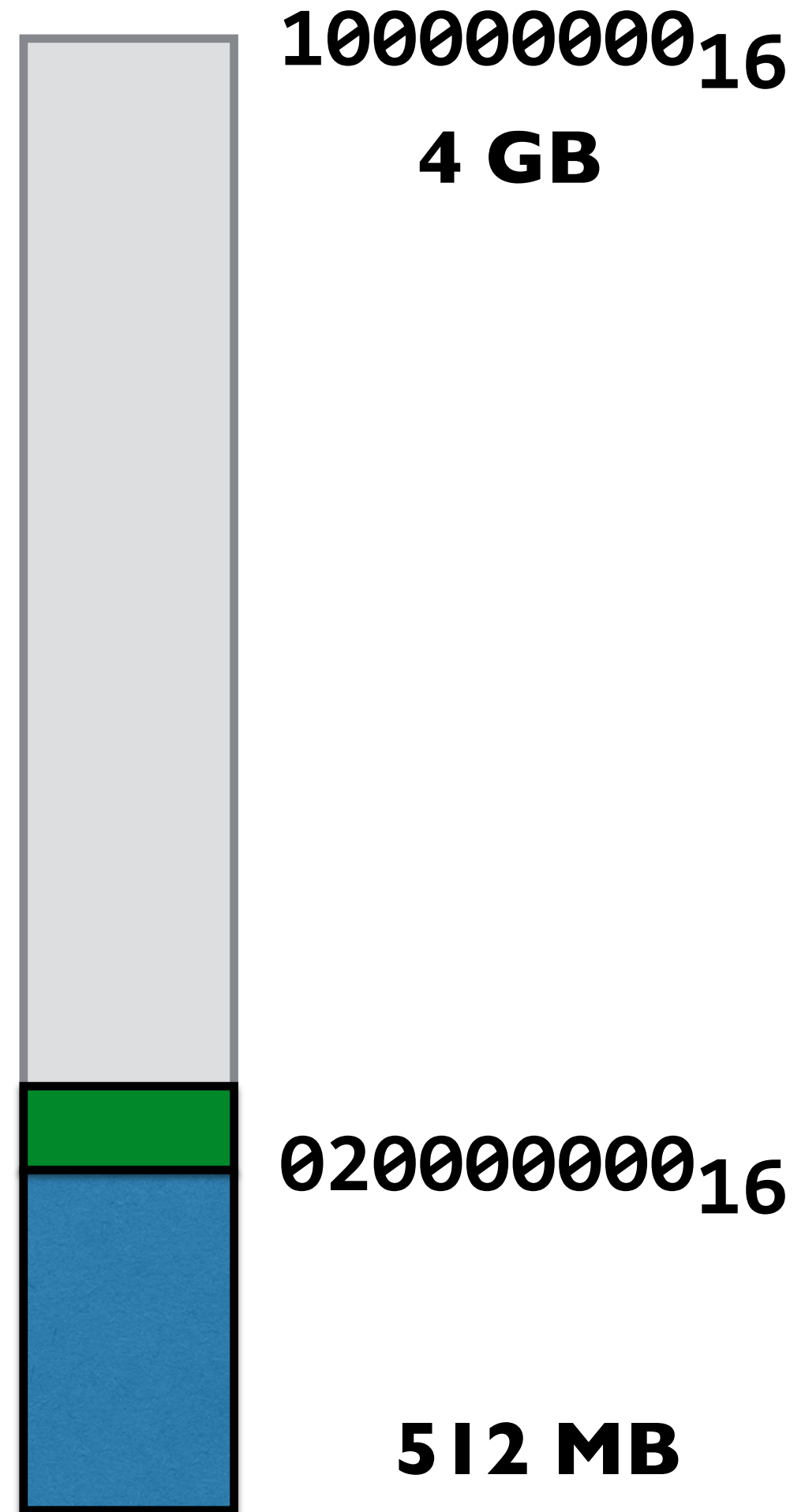 - lots of wiring — you need to work efficiently

Goals for today

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

Memory

Memory is a linear sequence of bytes

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

100000000_{16}

4 GB

020000000_{16}

512 MB

Memory as a linear sequence of indexed bytes

[8010]	
	20
	20
	00
[800c]	20
	e5
	80
	10
[8008]	00
	e3
	a0
	19
[8004]	02
[8003]	e5
[8002]	9f
[8001]	00
[8000]	04

Same memory, grouped into 4-byte words

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

Note little-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)

ASM and memory

At the assembly level, a 4-byte word could represent

- an address,

- an int,

- 4 characters

- an ARM instruction

The ldr/str instructions are agnostic to type: assembly has no type system to guide or restrict us on what we do with those words.

C pointer types

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

A pointer is a variable that holds an address.
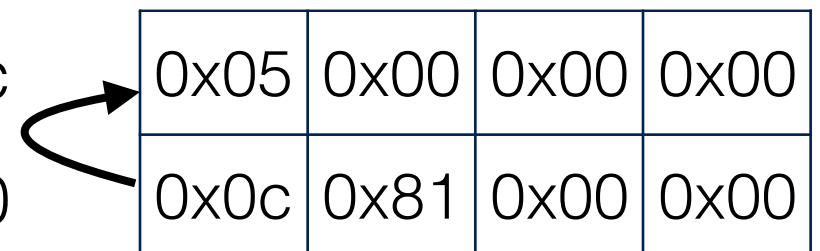
The "pointee" is the data stored at that address.

C code	address	memory

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

0x0000810c → | 0x05 | 0x00 | 0x00 | 0x00 |

0x00008110 → | 0x0c | 0x81 | 0x00 | 0x00 |

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

C Pointer Operations

```
int val = 5;
int* ptr = &val;
```
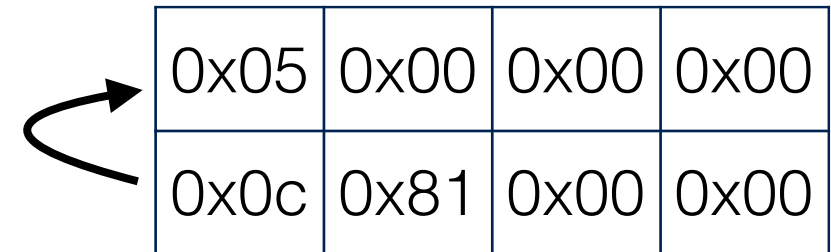
0x0000810c

0x00008110

0x05	0x00	0x00	0x00
0x0c	0x81	0x00	0x00

C Pointer Operations

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

0x0000810c

0x00008110

0x05	0x00	0x00	0x00
0x0c	0x81	0x00	0x00

```
*ptr = 7;
```

0x0000810c

0x00008110

0x07	0x00	0x00	0x00
0x0c	0x81	0x00	0x00

C Pointer Operations

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

0x0000810c

0x00008110

0x05	0x00	0x00	0x00
0x0c	0x81	0x00	0x00

```
*ptr = 7;
```

0x0000810c

0x00008110

0x07	0x00	0x00	0x00
0x0c	0x81	0x00	0x00

```
int** dptr = &ptr;
```

0x0000810c

0x00008110

0x00008114

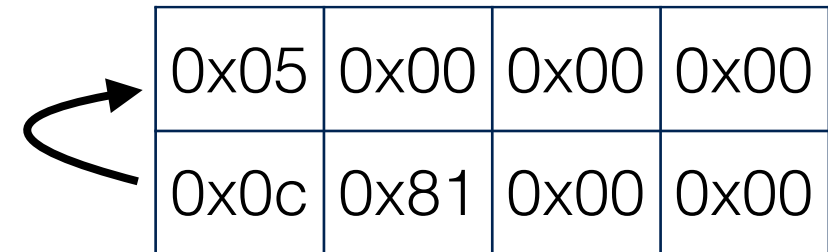| 0x07	0x00	0x00	0x00
0x0c	0x81	0x00	0x00
0x10	0x81	0x00	0x00

C Pointer Operations

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

0x0000810c

0x00008110

0x05	0x00	0x00	0x00
0x0c	0x81	0x00	0x00

```
*ptr = 7;
```

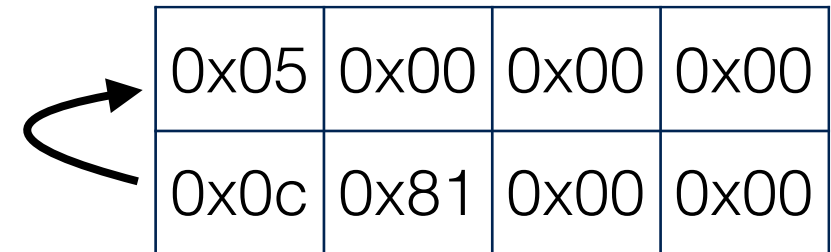0x0000810c

0x00008110

0x07	0x00	0x00	0x00
0x0c	0x81	0x00	0x00

```
int** dptr = &ptr;
```

0x0000810c

0x00008110

0x00008114

0x07	0x00	0x00	0x00
0x0c	0x81	0x00	0x00
0x10	0x81	0x00	0x00

```
*dptr = NULL;
```

0x0000810c

0x00008110

0x00008114

0x07	0x00	0x00	0x00
0x00	0x00	0x00	0x00
0x10	0x81	0x00	0x00

C Pointer Operations

```
char a = 'a';
char b = 'b'
char* ptr = &b;
```

0x0000810c

0x00008110

0x61	0x62		
'a'	'b'	0x00	0x00
0x0d	0x81	0x00	0x00

C Pointer Operations

```
char a = 'a';
char b = 'b';
char* ptr = &b;

*ptr = 'c';
```

0x0000810c

0x00008110

0x0000810c

0x00008110

0x61 0x62

'a'	'b'	0x00	0x00
0x0d	0x81	0x00	0x00

0x63

'a'	'c'	0x00	0x00
0x0d	0x81	0x00	0x00

C Pointer Operations

```
char a = 'a';
char b = 'b'
char* ptr = &b;
```

0x0000810c

0x00008110

'a'	'b'	0x00	0x00
0x0d	0x81	0x00	0x00

```
*ptr = 'c';
```

0x0000810c

0x00008110

'a'	'c'	0x00	0x00
0x0d	0x81	0x00	0x00

```
char** dptr = &ptr;
```

0x0000810c

0x00008110

0x00008114

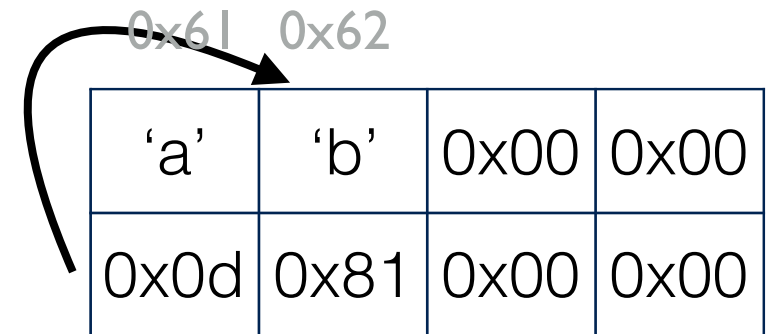| 'a'	'c'	0x00	0x00
0x0d	0x81	0x00	0x00
0x10	0x81	0x00	0x00

C Pointer Operations

```
char a = 'a';
char b = 'b'
char* ptr = &b;
```

0x0000810c

0x00008110

0x61   0x62

'a'	'b'	0x00	0x00
0x0d	0x81	0x00	0x00

```
*ptr = 'c';
```

0x0000810c

0x00008110

0x63

'a'	'c'	0x00	0x00
0x0c	0x81	0x00	0x00

```
char** dptr = &ptr;
```

0x0000810c

0x00008110

0x00008114

'a'	'c'	0x00	0x00
0x0d	0x81	0x00	0x00
0x10	0x81	0x00	0x00

```
*dptr = NULL;
```

0x0000810c

0x00008110

0x00008114

'a'	'c'	0x00	0x00
0x00	0x00	0x00	0x00
0x10	0x81	0x00	0x00

C pointer types

C has a *type system:* tracks the type of each variable.

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

C arrays

An array allocates multiple instances of a type contiguously in memory

```
char ab[2];
ab[0] = 'a';
ab[1] = 'b';
```

0x0000810c

0x61	0x62		
'a'	'b'		

```
int ab[2];
ab[0] = 'a';
ab[1] = 9;
```

0x0000810c

0x00008110

0x61			
'a'	0x00	0x00	0x00
0x09	0x00	0x00	0x00

Arrays and Pointers

You can assign an array to a pointer

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

Incrementing pointers advances address by size of type

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

What does the assembly look like?
What if ab is a char[2] and ptr is a char*?

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?
```

Pointers 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?
```

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
```

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)

```
for (char* ptr = &ch; *ptr != 0; ++ptr) {}
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)

```
for (char* ptr = &ch; *ptr != 0; ptr++) {}
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-strings

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

// which assignments are valid?
1  ptr = s;
2  ptr = arr;
3  ptr = buf;
4  arr = ptr;
5  buf = oldschool;
```

```
\0
64
63
61
6c
65
4c
```

```
??\06463
616c654c
```

What does a typecast **actually do**?

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

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

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

Power of Types and Pointers

```c
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];
};
```

| 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

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

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?
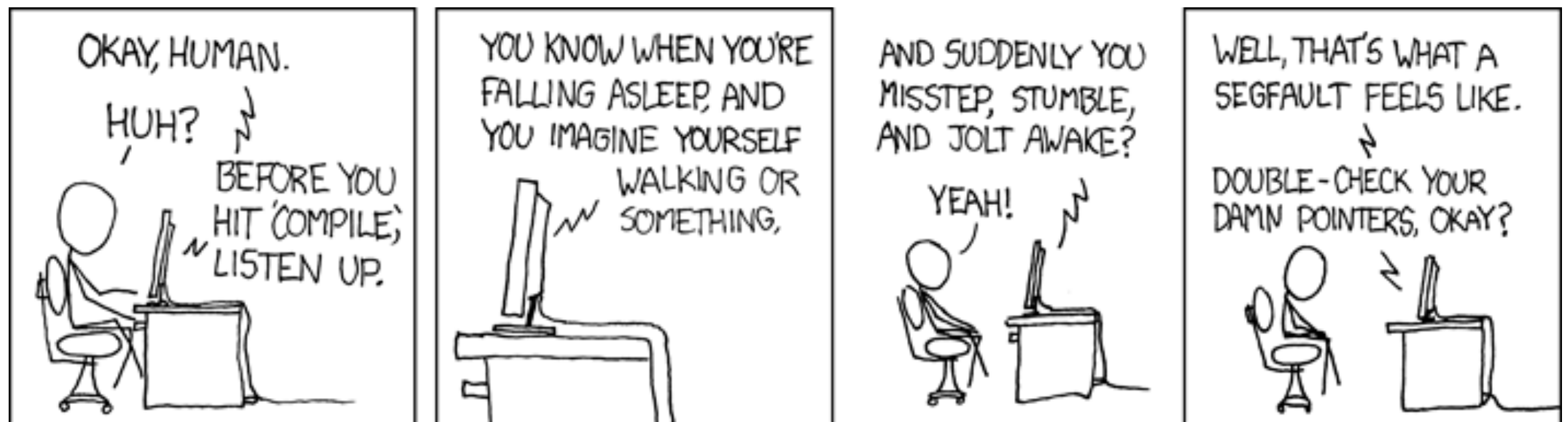
C vs. Assembly

```
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

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 can use its knowledge of when it is read/written to optimize accesses as long as it keeps the same externally visible 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.

What is 'bare metal'?

The default build process for C assumes a *hosted* environment. It provides standard libraries, all the stuff that happens before `main`.

To build bare-metal, our makefile disables these defaults; we must supply our own versions when needed.

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

Makefile settings

Compile freestanding

```
CFLAGS =-ffreestanding
```

Link without standard libs and start files

```
LDFLAGS = -nostdlib
```

Link with gcc to support division (violates

```
LDLIBS = -lgcc
```

Must supply own replacement for libs/start

That's where the fun is…!

Pseudo-instruction!
(assemblers are helpful)

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
```

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
```

Quiz: what does this program do

… on linux?

… on MacOS?

… on your Pi?

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

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