

This week

- Assignment 1 due Tuesday: you'll have proved your bare-metal mettle!
- Lab 2 prep
 - do pre-lab reading!
 - bring your tools if you have them



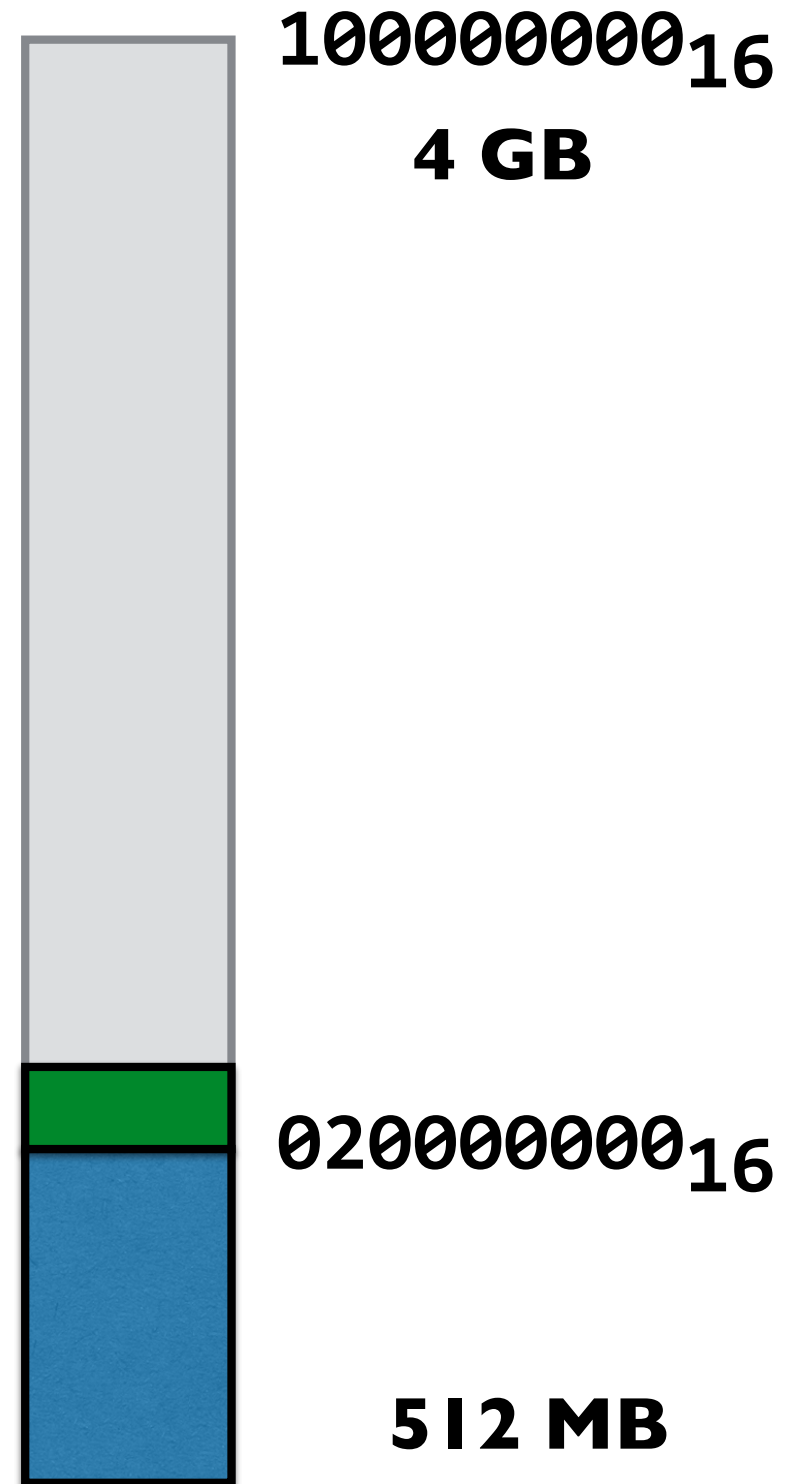
Goals for today

- 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)



Accessing memory in assembly

ldr copies 4 bytes from memory address to register

str copies 4 bytes from register to memory address

The memory address could be:

- the location of a global or local variable or
- the location of program instructions or
- a memory-mapped peripheral or
- an unused/invalid location or ...

The 4 bytes of data being copied could be:

- an address or
- an 32-bit integer or
- 4 characters or
- an ARM instruction, or...

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

```
ldr r0, FSEL2  
mov r1, #1  
str r1, [r0]
```

```
ldr r0, SET0  
mov r1, #(1<<20)  
str r1, [r0]
```

And assembly code doesn't care

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

Assembly has no type system to guide or restrict us on what we do with those words.

Keeping track of what's what in assembly is *hard* and very bug-prone.

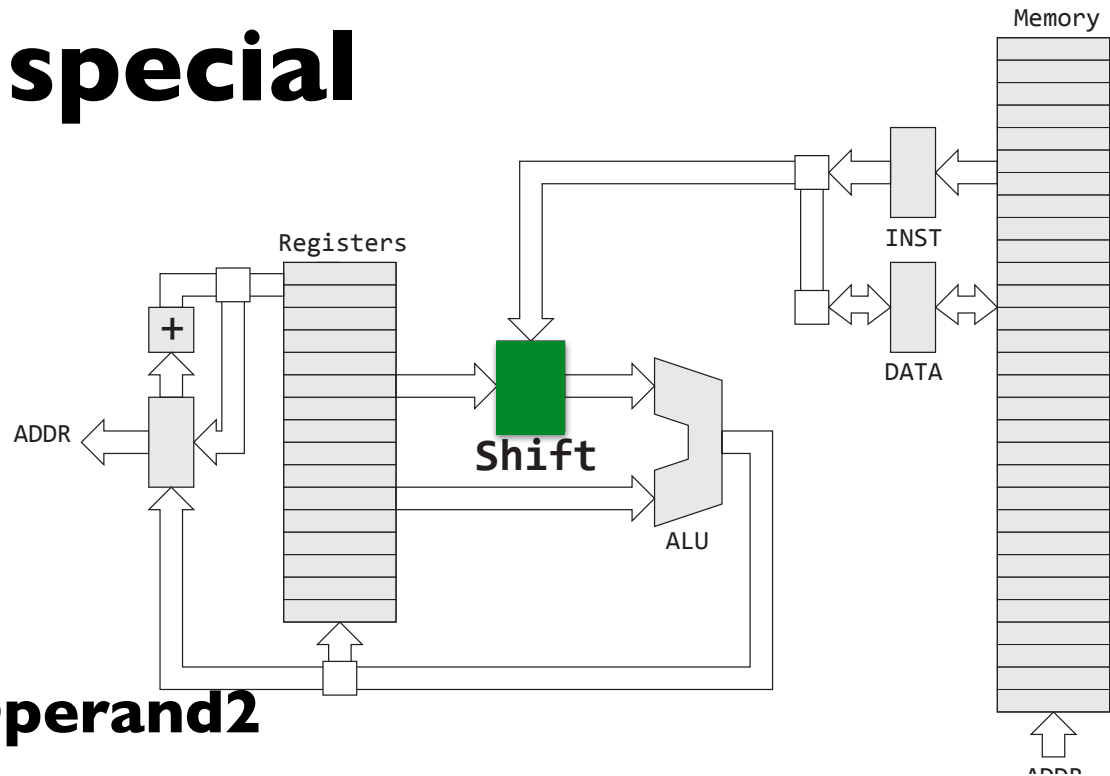
Funny program

pc is the register containing the address of the current instruction (processor updates it on each execution, changes it on branch instructions)

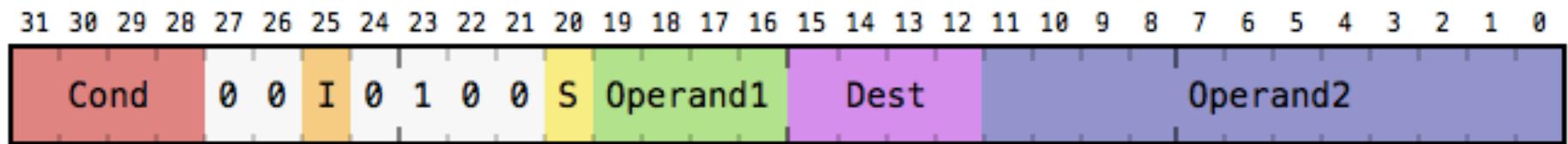
What does this program do?

```
ldr r1, [pc - 4]  
add r1, r1, #1  
str r1, [pc - 12]
```

Operand 2 is special



Dest = Operand1 op Operand2



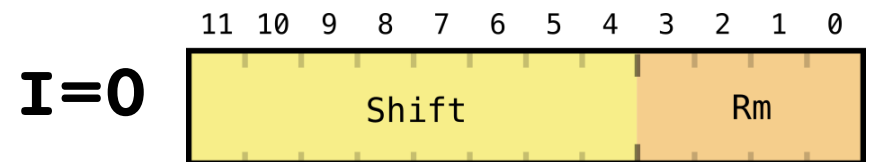
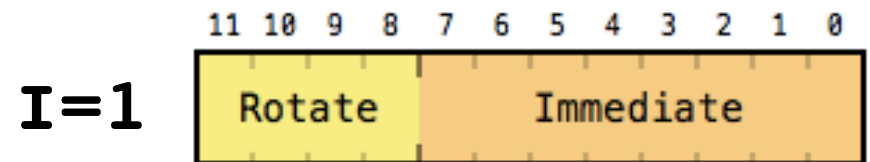
add r0, r1, #0xf000

sub r0, r1, #6

rsb r0, r1, #6

add r0, r1, r2, lsl #3

mov r1, r2, ror #7



lsl, lsr, asr, ror

Funny program

pc is the register containing the address of the current instruction (processor updates it on each execution, changes it on branch instructions)

What does this program do?

		e2	81	10	01
ldr r1, [pc - 4]	+1	=	e2	81	10 02
add r1, r1, #1	+2	=	e2	81	10 04
str r1, [pc - 12]	+4	=	e2	81	10 08
					...
	+ 128	=	e2	81	11 00

Funny program

pc is the register containing the address of the current instruction (processor updates it on each execution, changes it on branch instructions)

What does this program do?

	e2 81 10 01
ldr r1, [pc - 4]	+1 = e2 81 10 02
add r1, r1, #1	+2 = e2 81 10 04
str r1, [pc - 12]	+4 = e2 81 10 08
	...
	+ 128 = e2 81 11 00

Operating on addresses is extremely powerful!
We need some safety rails.

C pointer vocabulary

An **address** is a memory location. Representation is unsigned 32-bit int.

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

The “**pointee**” is the data stored at that address.

* is the **dereference** operator, & is **address-of**.

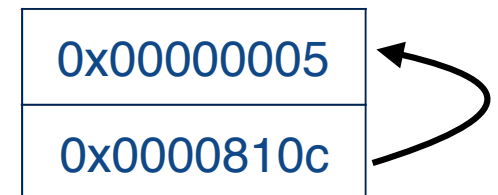
C code

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

val [810c]

ptr [8108]

Memory



What do C pointers buy us?

- Access specific memory by address, e.g. FSEL2
- Allow us to specify not only an address, but also what we expect to be stored at that address: the data type
 - `int*` vs `char*` vs `key_event_t*`
- Access data by its offset relative to other nearby data (array elements, struct fields)
 - Storing related data in related locations organizes use of memory
- Efficiently refer to shared data, avoid redundancy/duplication
- Build flexible, dynamic data structures at runtime

CULTURE FACT:

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

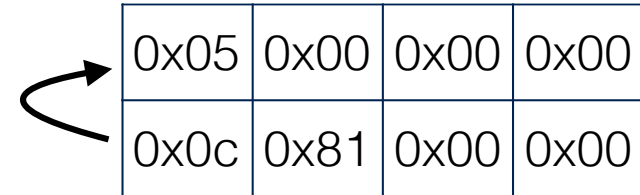


C Pointer Operations

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

0x0000810c

0x00008110



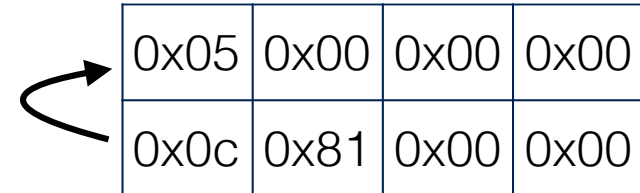
0x05	0x00	0x00	0x00
0x0c	0x81	0x00	0x00

C Pointer Operations

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

0x0000810c

0x00008110

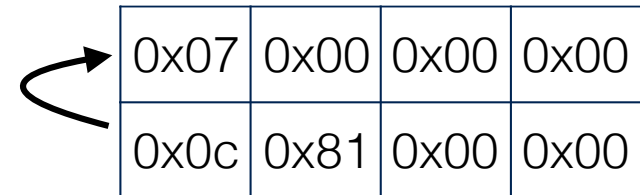


0x05	0x00	0x00	0x00
0x0c	0x81	0x00	0x00

```
*ptr = 7;
```

0x0000810c

0x00008110



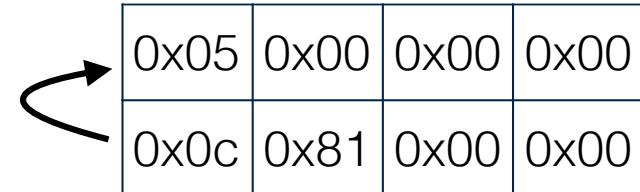
0x07	0x00	0x00	0x00
0x0c	0x81	0x00	0x00

C Pointer Operations

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

0x0000810c

0x00008110

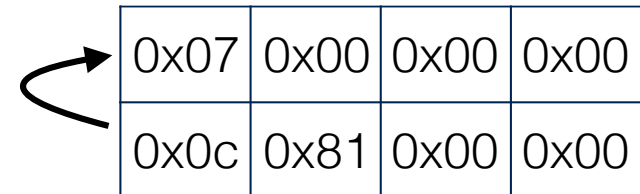


0x05	0x00	0x00	0x00
0x0c	0x81	0x00	0x00

```
*ptr = 7;
```

0x0000810c

0x00008110



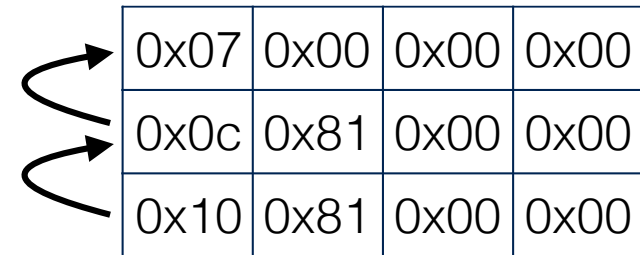
0x07	0x00	0x00	0x00
0x0c	0x81	0x00	0x00

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

0x0000810c

0x00008110

0x00008114



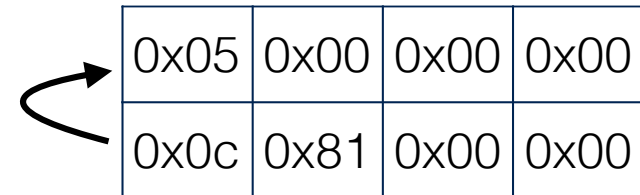
0x07	0x00	0x00	0x00
0x0c	0x81	0x00	0x00
0x10	0x81	0x00	0x00

C Pointer Operations

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

0x0000810c

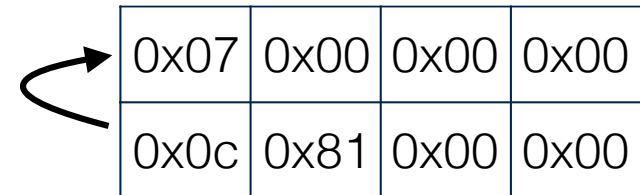
0x00008110



```
*ptr = 7;
```

0x0000810c

0x00008110

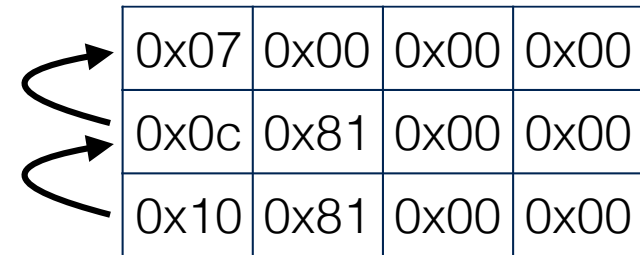


```
int** dptr = &ptr;
```

0x0000810c

0x00008110

0x00008114

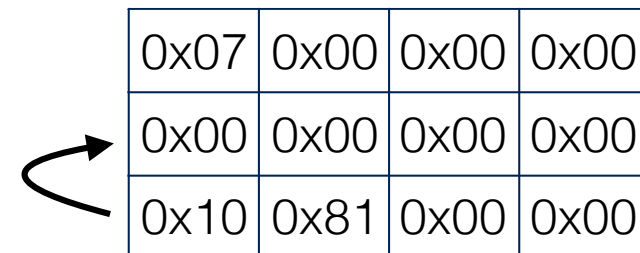


```
*dptr = NULL;
```

0x0000810c

0x00008110

0x00008114

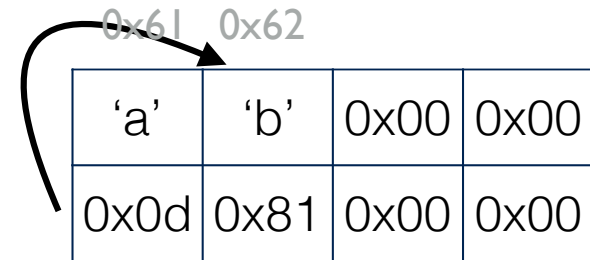


C Pointer Operations

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

0x00000810c

0x000008110



C Pointer Operations

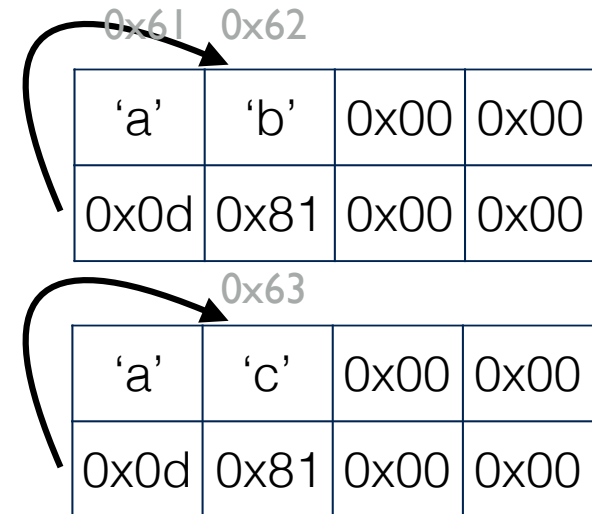
```
char a = 'a';  
char b = 'b';  
char* ptr = &b;  
  
*ptr = 'c';
```

0x00000810c

0x000008110

0x00000810c

0x000008110



C Pointer Operations

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

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

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

0x0000810c

0x00008110

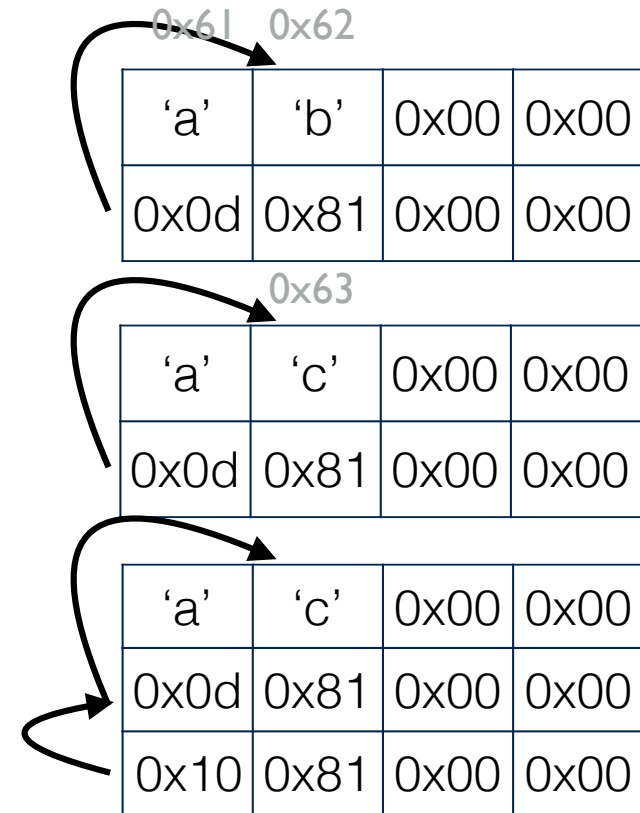
0x0000810c

0x00008110

0x0000810c

0x00008110

0x00008114



C Pointer Operations

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

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

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

```
*dptr = NULL;
```

0x0000810c

0x00008110

0x0000810c

0x00008110

0x0000810c

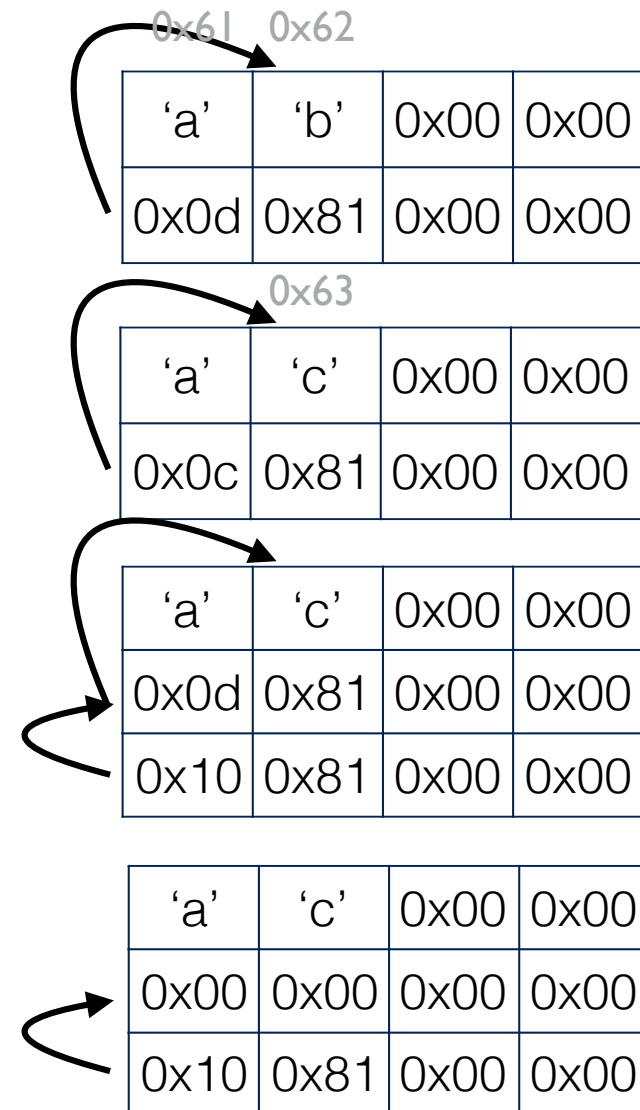
0x00008110

0x00008114

0x0000810c

0x00008110

0x00008114



Pointer Quiz: & *

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

```
p = &n;
```

```
*p = n;           // 1. same as prev line?
```

```
q = p;
```

```
*q = *p;          // 2. same as prev line?
```

```
p = &m, q = &n;
```

```
*p = *q;
```

```
m = n;           // 3. same as prev line?
```

C pointer types

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

Operations have 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 Arithmetic

Incrementing pointers advances address by size of type.

```
struct point {  
    int x; // 32 bits, 4 bytes  
    int y; // 32 bits, 4 bytes  
};  
struct point points[100];  
struct point* ptr = points;  
ptr = ptr + 1; // now points to points[1]
```

Suppose points is at address 0x100. What is the value of ptr after the last line of code?

Pointers and arrays

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

```
p = arr;
```

```
p = &arr[0];           // same as prev line
```

```
arr = p;                // ILLEGAL, why?
```

```
*p = 3;
```

```
p[0] = 3;              // same as prev line
```

```
n = *(arr + 1);
```

```
n = arr[1];            // same as prev line
```

Address arithmetic

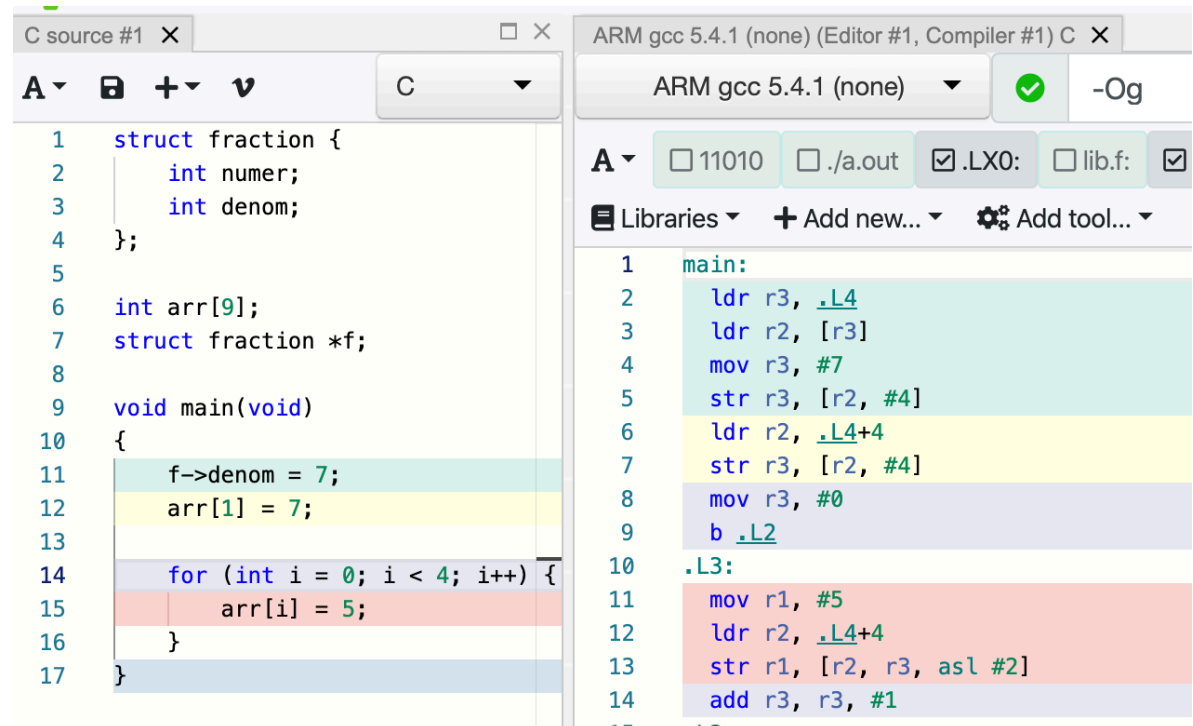
Fancy ARM addressing modes

```
ldr r0, [r1, #4]           // constant displacement
ldr r0, [r1, r2]           // variable displacement
ldr r0, [r1, r2, asl #3]   // scaled index displacement
```

(Even fancier variants add pre/post update to move pointer along)

Q: How do these relate to accessing data structures in C?

Try CompilerExplorer to find out!



The screenshot shows the Compiler Explorer interface. On the left, the C source code is displayed in a file named 'C source #1'. It defines a 'fraction' struct with 'numer' and 'denom' fields, an array 'arr' of size 9, and a 'main' function that initializes 'denom' to 7, sets 'arr[1]' to 7, and then iterates over 'arr' from index 0 to 3, setting each element to 5. On the right, the ARM assembly output is shown for 'ARM gcc 5.4.1 (none)'. The assembly includes instructions for loading and storing values, moving registers, and branching, corresponding to the C code's logic. The assembly is color-coded to match the C code's structure.

```
C source #1 X
1 struct fraction {
2     int numer;
3     int denom;
4 };
5
6 int arr[9];
7 struct fraction *f;
8
9 void main(void)
10 {
11     f->denom = 7;
12     arr[1] = 7;
13
14     for (int i = 0; i < 4; i++) {
15         arr[i] = 5;
16     }
17 }
```

```
ARM gcc 5.4.1 (none) (Editor #1, Compiler #1) C X
ARM gcc 5.4.1 (none) -Og
11010 .a.out .LX0: lib.f:
Libraries + Add new... Add tool...
1 main:
2     ldr r3, .L4
3     ldr r2, [r3]
4     mov r3, #7
5     str r3, [r2, #4]
6     ldr r2, .L4+4
7     str r3, [r2, #4]
8     mov r3, #0
9     b .L2
10 .L3:
11     mov r1, #5
12     ldr r2, .L4+4
13     str r1, [r2, r3, asl #2]
14     add r3, r3, #1
```

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

```
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

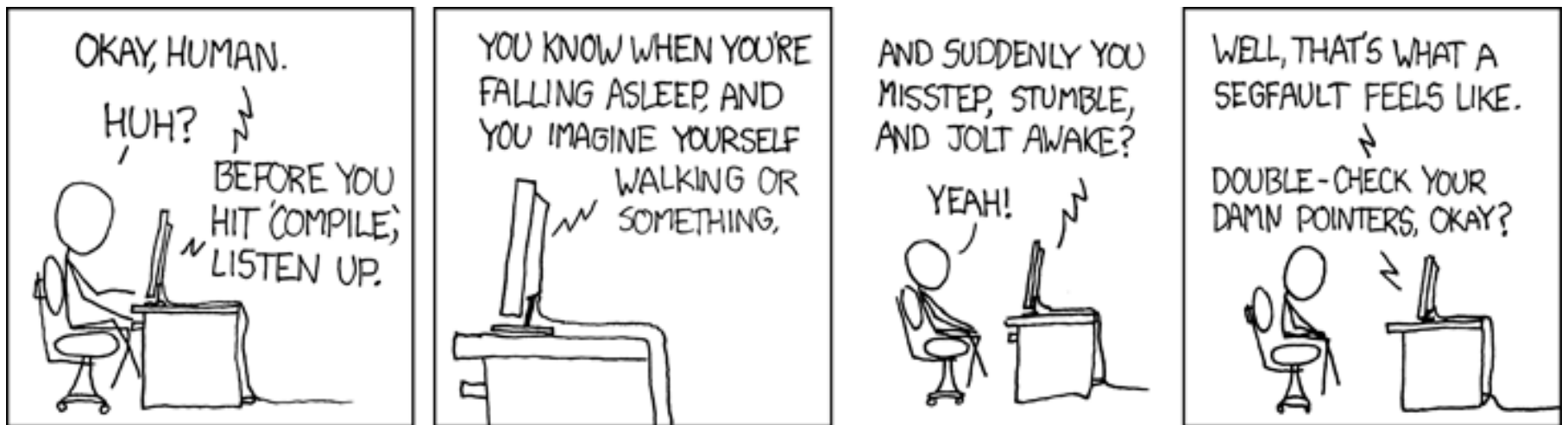
```
volatile struct gpio *gpio = (struct gpio *)0x20200000;  
gpio->fsel[0] = ...
```

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?

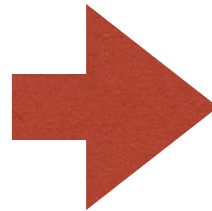


C vs. Assembly


```

.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

```



C?

Let's do it!

```

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

```

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,
you get what you see.

For C source, you may need to look
at what compiler has generated to be
sure of what you're getting.

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

When Your C Compiler Is Too Smart For Its Own Good

(or, why every systems programmer should be able to
read 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 ordinary memory.

For example: Writing a 1 bit into SET register sets output to 1; writing a 0 bit into SET register has no effect. Writing a 1 bit into CLR sets the output to 0; writing a 0 bit into CLR has no effect. To read the current value, access the LEV (level) register. So writing to SET can change the value of LEV, a different memory address!

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

volatile

Ordinarily, the compiler uses its knowledge of reads/writes to optimize while keeping the same externally visible behavior.

However, for a variable that can be read/written externally in a way the C compiler can't know (by another process, by hardware), these optimizations may not be valid.

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

(If ever in doubt about what the compiler has done, use tools to review generated assembly and see for yourself...!)