Check in!

From Assembly to C (and back again)

Goals for today

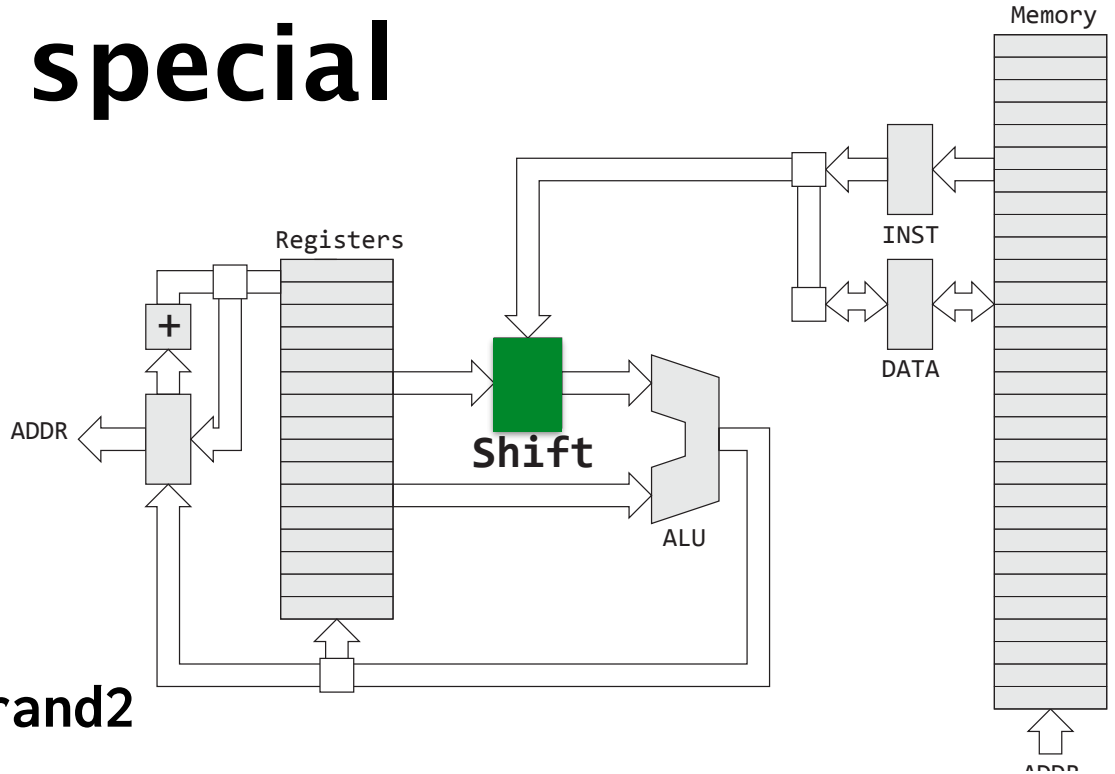
ARM condition codes, branch instructions

C language as “high-level” assembly

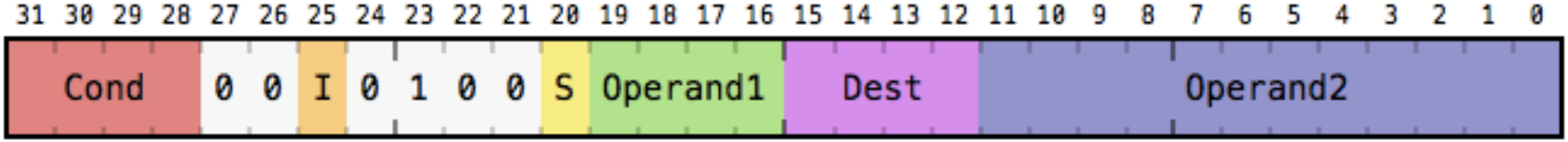
Relationship of C to asm

What does a compiler do?

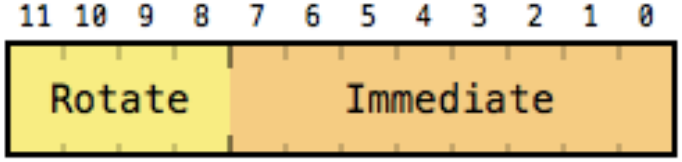
Operand 2 is special



Dest = Operand1 op Operand2




```
add r0, r1, #0x3f000
sub r0, r1, #7
rsb r0, r1, #7
add r0, r1, r2, lsl #3
mov r1, r2, ror #7
```



lsl, lsr, asr, ror

Control flow

Instructions stored in contiguous memory

Register **pc** (**r15**) tracks address in memory where instructions are being read

Default is "straight-line" code: next instruction to execute is at next higher memory address (**pc = pc + 4**)

Change **pc** to change which instruction is next, **branch** instruction has effect of **pc = target**

```
b target
```

Above branch is *unconditional* (always taken), alternative is predicated on state of "condition codes"

Condition Codes

Z result was 0

N result was < 0

C operation generated carry

V operation had arithmetic overflow

(More on carry and overflow in later lecture…)

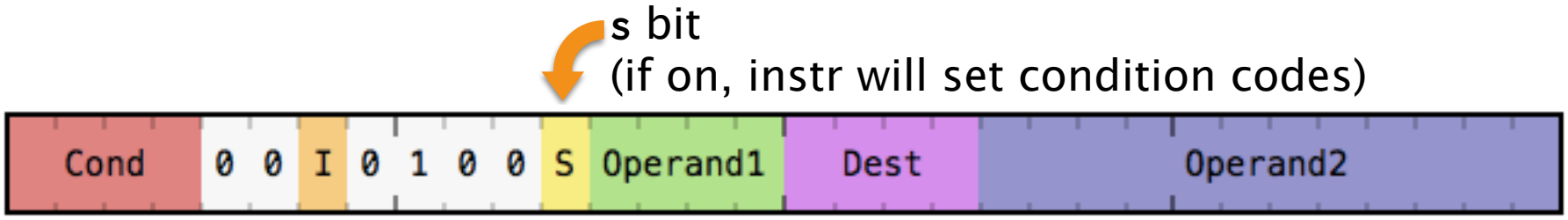
Which instructions set/clear codes?

`cmp` (like a **`sub`**, but discards result)

`tst` (like an **`and`**, but discards result)

Data processing instruction suffixed with **`s`**:

`adds movs orrs lsrs ...`



Code	Suffix	Description	Flags
0000	EQ	Equal / equals zero	Z
0001	NE	Not equal	!Z
0010	CS / HS	Carry set / unsigned higher or same	C
0011	CC / LO	Carry clear / unsigned lower	!C
0100	MI	Minus / negative	N
0101	PL	Plus / positive or zero	!N
0110	VS	Overflow	V
0111	VC	No overflow	!V
1000	HI	Unsigned higher	C and !Z
1001	LS	Unsigned lower or same	!C or Z
1010	GE	Signed greater than or equal	N == V
1011	LT	Signed less than	N != V
1100	GT	Signed greater than	!Z and (N == V)
1101	LE	Signed less than or equal	Z or (N != V)
1110	AL	Always (default)	any

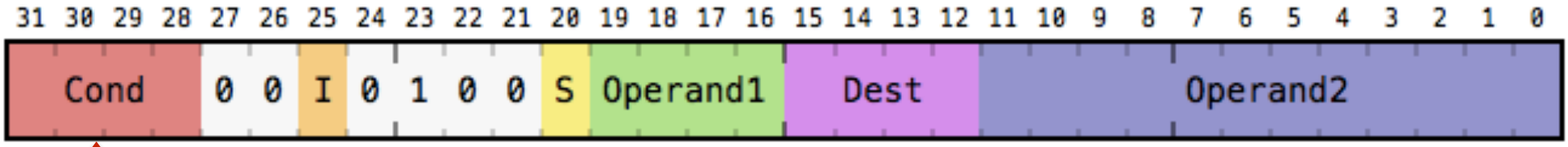
Branch instructions

```
b target
bne target
bmi target
bge target
...
```

branch instruction reads condition codes, as set by a previous instruction

If specific condition(s) satisfied, branch is taken and **pc = target** otherwise falls through and **pc = pc + 4**

Condition execution



The ability to conditionally execute is built into all ARM instructions! (not just **branch**…)

```
addeq r0, r0, #3
submi r1, r2, r3
```

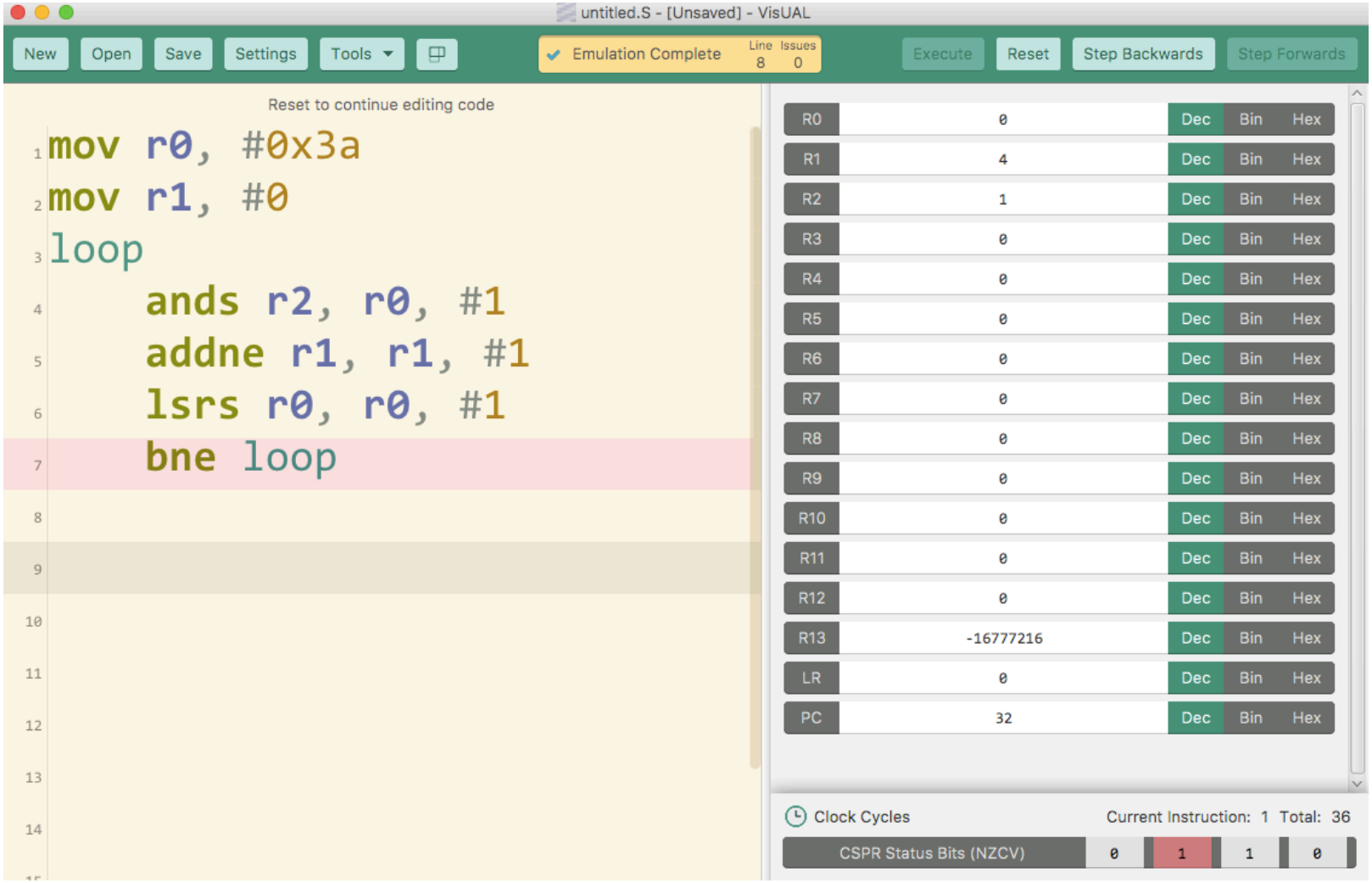
Q: Given our earlier foray into machine-encoded instructions, what do you suspect is the condition represented by `0xe`*?*

Challenge for you all:

Write an assembly program to count the "on" bits in a given numeric value

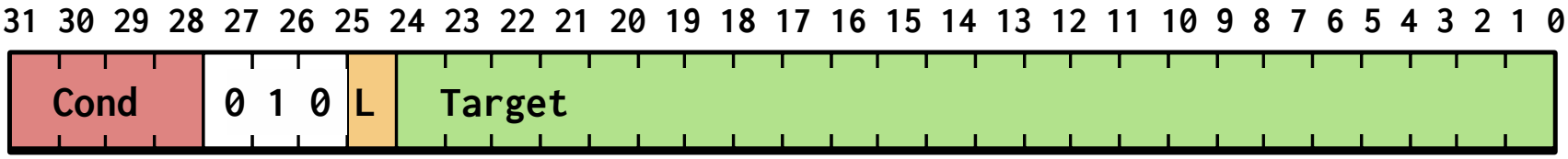
```
mov r0, #val
mov r1, #0

// r0 holds input value to start
// use r1 to store count of on bits in r0
```

VisUAL ARM Emulator



https://salmanarif.bitbucket.io/visual/

Branch instruction encoding



b (bal) branch always

1110 1010 tttt tttt tttt tttt tttt tttt

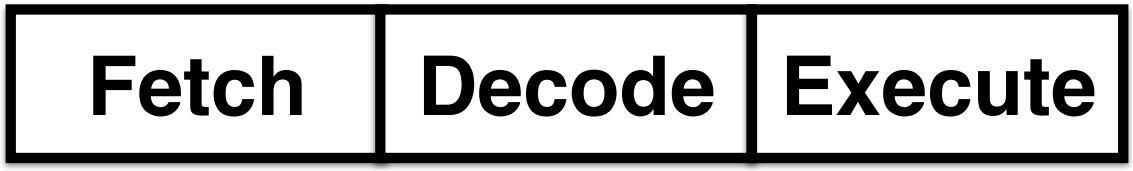
beq branch if zero CC set

0000 1010 tttt tttt tttt tttt tttt tttt

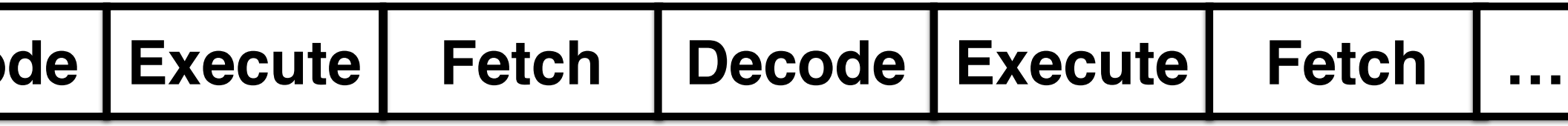
branch target is PC-relative offset
green bits encode offset, counted in 4-byte words

Q: How far can this reach?

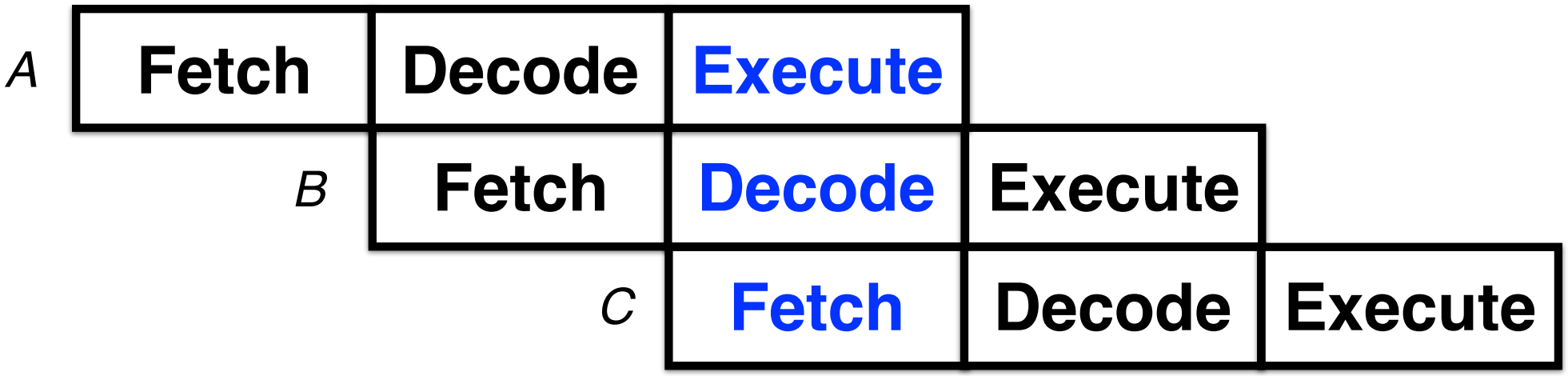
3 steps per instruction



3 instructions takes 9 steps in sequence

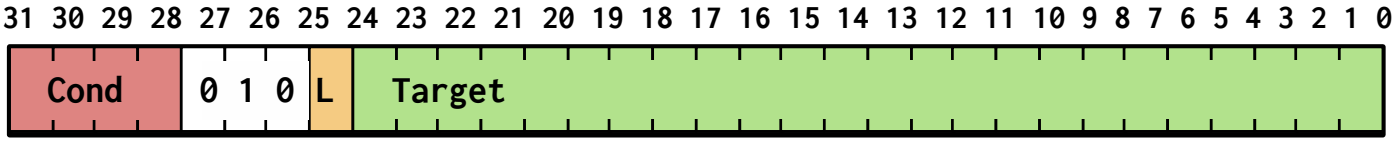


To speed things up, steps are overlapped ("pipelined")



During cycle that instruction *A* is executing, PC has advanced twice, holds address of instruction being fetched (*C*). This is 2 instructions past *A* (PC+8)

```
18: e3130001    tst r3, #1
1c: 1a000003    bne 30
20: e28300dc    add r0, r3, #107
24: e2433004    sub r3, r3, #4
28: e0800003    add r0, r0, r3
2c: e1a00003    mov r0, r3
30: e2833001    add r3, r3, #1
```




```
// 1a          branch if not equal
// 000003      target, expressed PC-relative as
               count of 4-byte words
// this offset is 8 + 3*4 = 20 bytes ahead
```

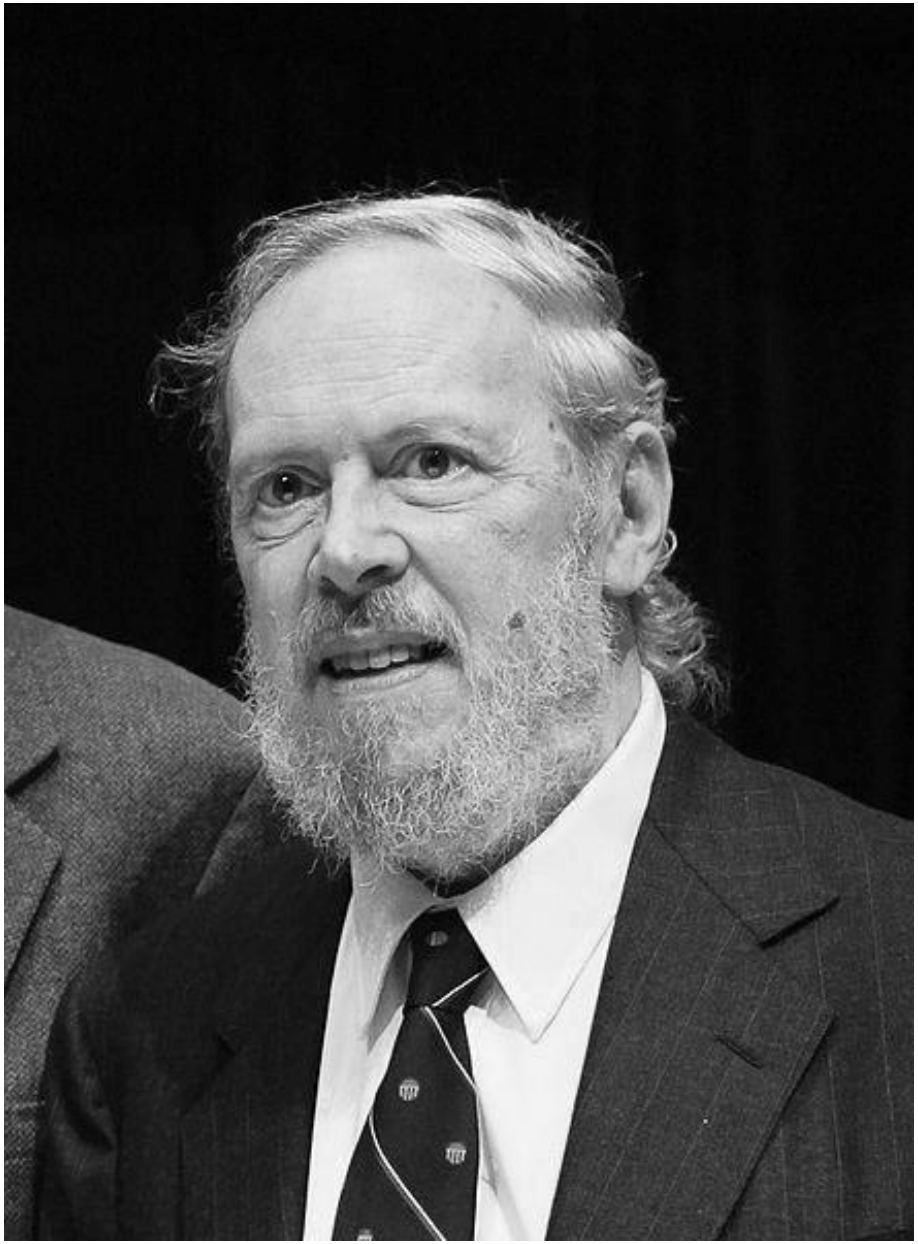
ISA design is an art form!

Some neat things about ARM design

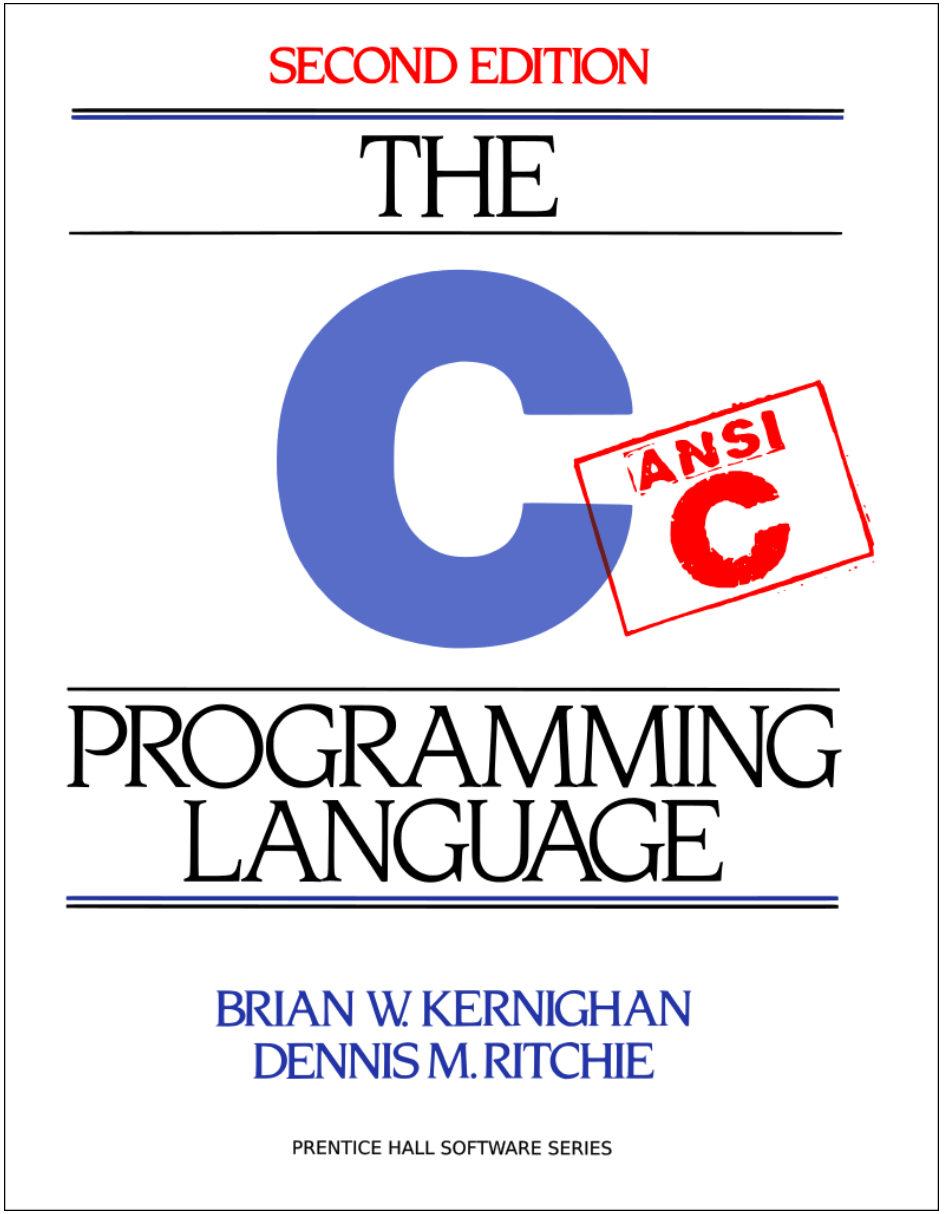
- Commonalities across operations
  - Register vs. immediate operands
  - Use of barrel shifter
  - All registers treated same (with a few caveats)
  - Predicated execution
  - Set condition code (or not)

Orthogonality leads to composability


100 FEET BELOW
SEA LEVEL



Dennis Ritchie



The C Programming Language

"C is quirky, flawed, and an enormous success"
— *Dennis Ritchie*

"C gives the programmer what the programmer wants; few restrictions, few complaints"
— *Herbert Schildt*

"C: A language that combines all the elegance and power of assembly language with all the readability and maintainability of assembly language"
— *Unknown*

Ken Thompson built UNIX using C

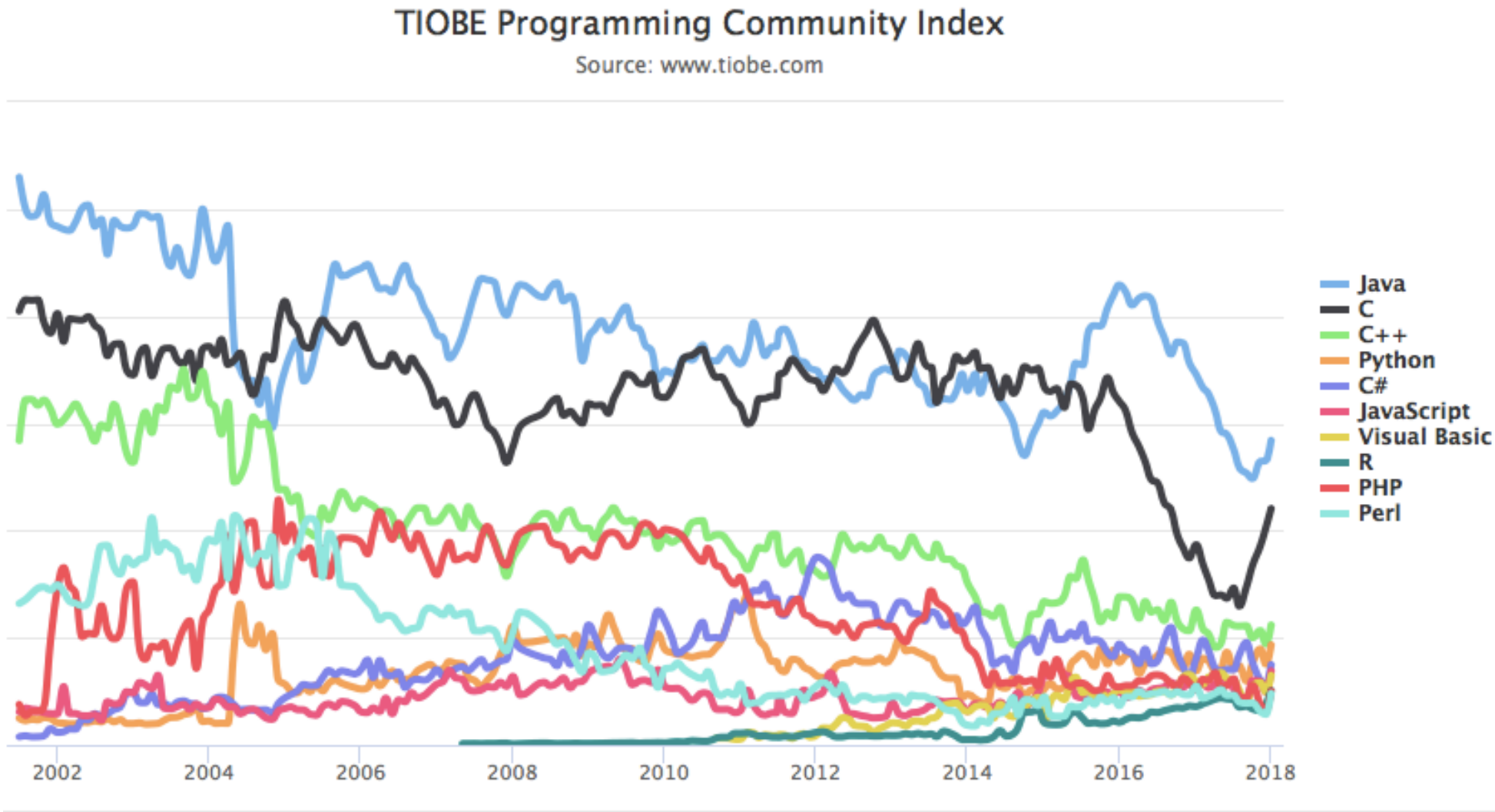
http://cm.bell-labs.com/cm/cs/who/dmr/picture.html

C is the language of choice for systems programmers

This is not coincidence!

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

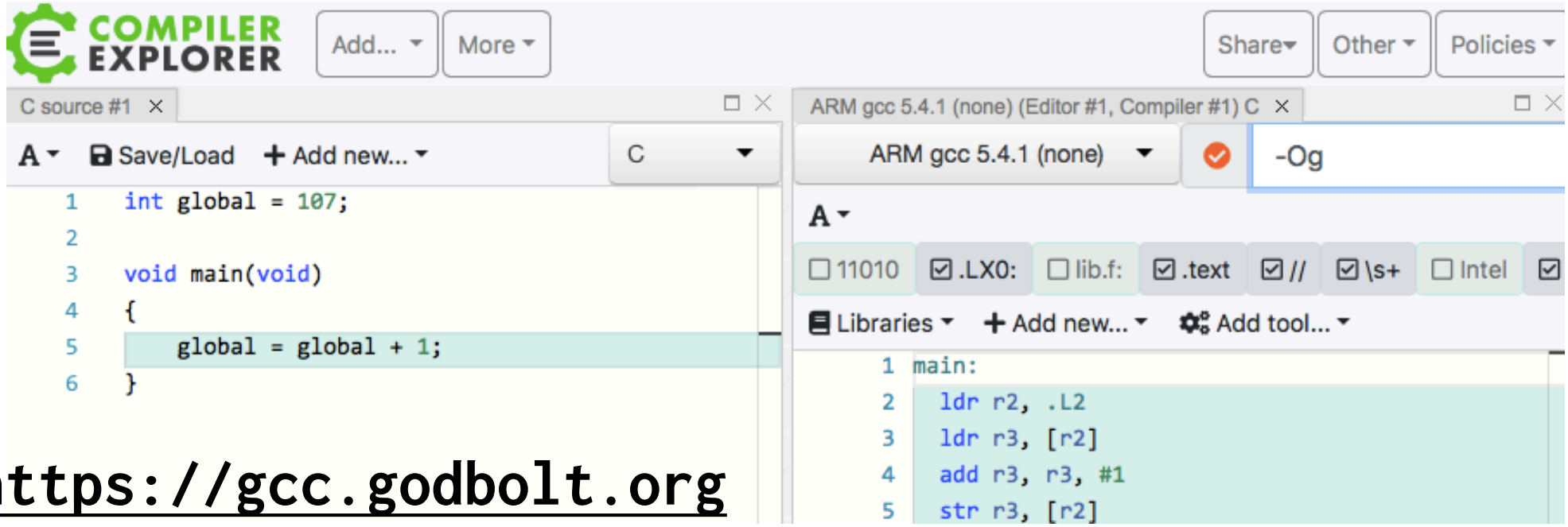
> "BCPL, B, and C family of languages are particularly oriented towards system programming, are small and compactly described, and are amenable to translation by simple compilers. They are "close to the machine" in that the abstractions they introduce are readily grounded in the concrete data types and operations supplied by conventional computers, and they rely on library routines for input-output and other interactions with an operating system. … At the same time, their abstractions lie at a sufficiently high level that, with care, portability between machines can be achieved."
>
> *- Dennis Ritchie*



Programming language popularity over time

Matt Godbolt's **Compiler Explorer**

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

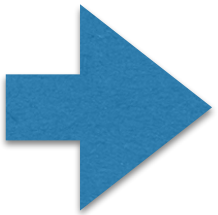


https://gcc.godbolt.org

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

```
.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!

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 compiler has 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

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 might happen if the C compiler makes assumptions about 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.