

Goals for today



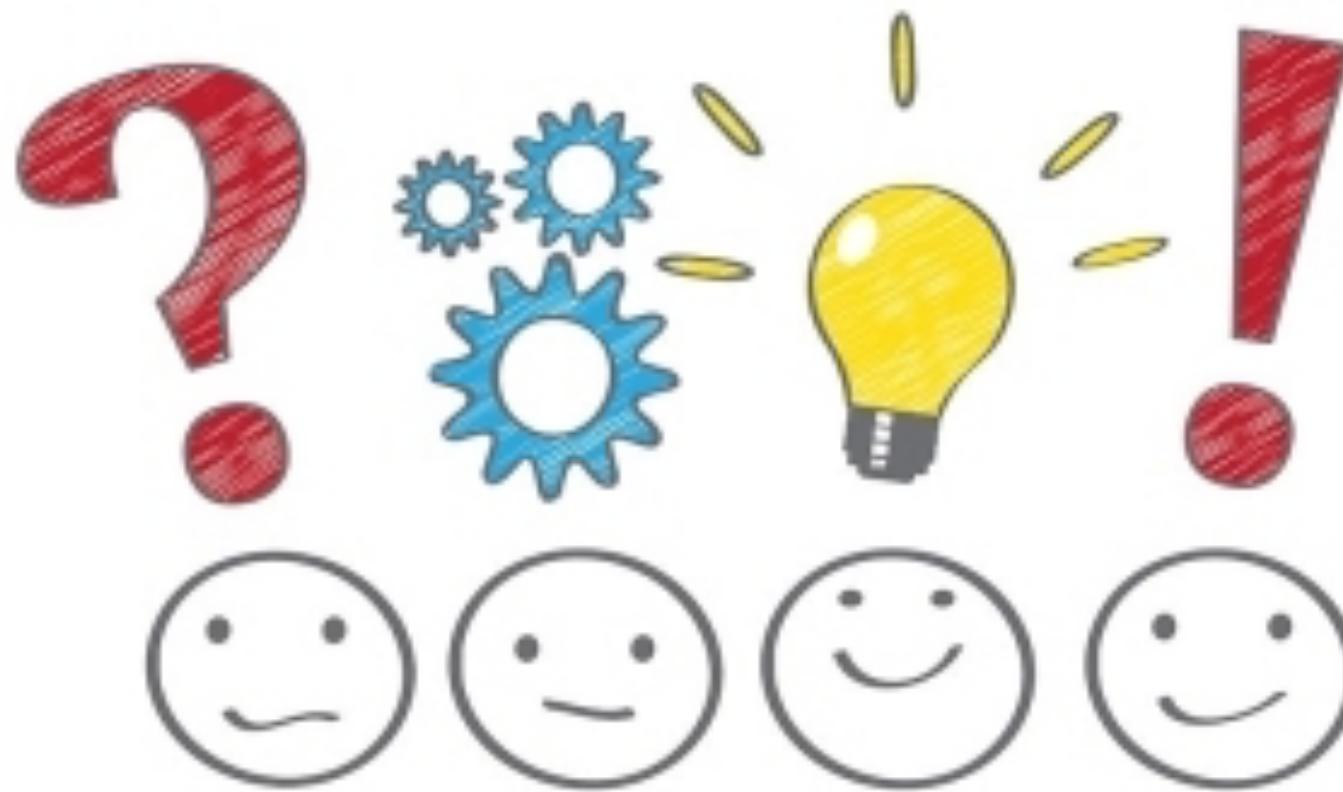
ARM condition codes, branch instructions

C language as “high-level” assembly

Relationship of C to asm

What does a compiler do?

Check in!



Control flow

Instructions stored in contiguous memory

Register pc (r15) uses to track where in memory to read instructions

"Straight-line" code: next instruction to execute is at next highest memory address

Redirect control flow by changing pc, typically via branch operation

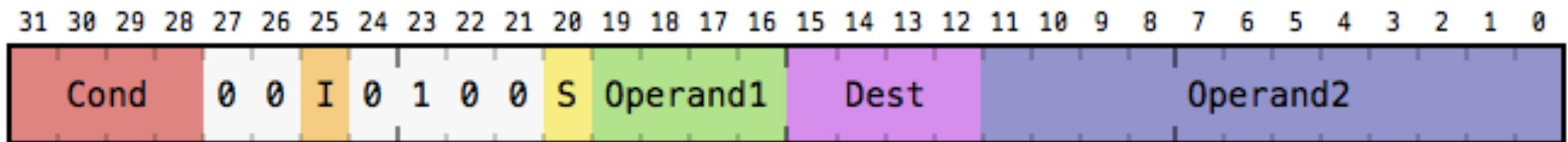
b target

Branch can be conditional as well as unconditional (predicted execution)

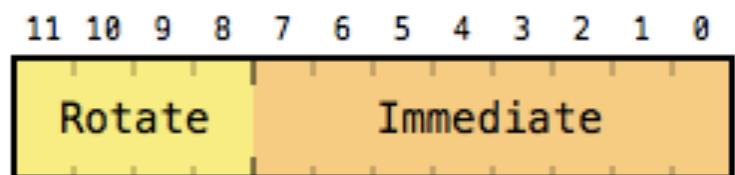
Data processing operations

`dst = operand1 op operand2`

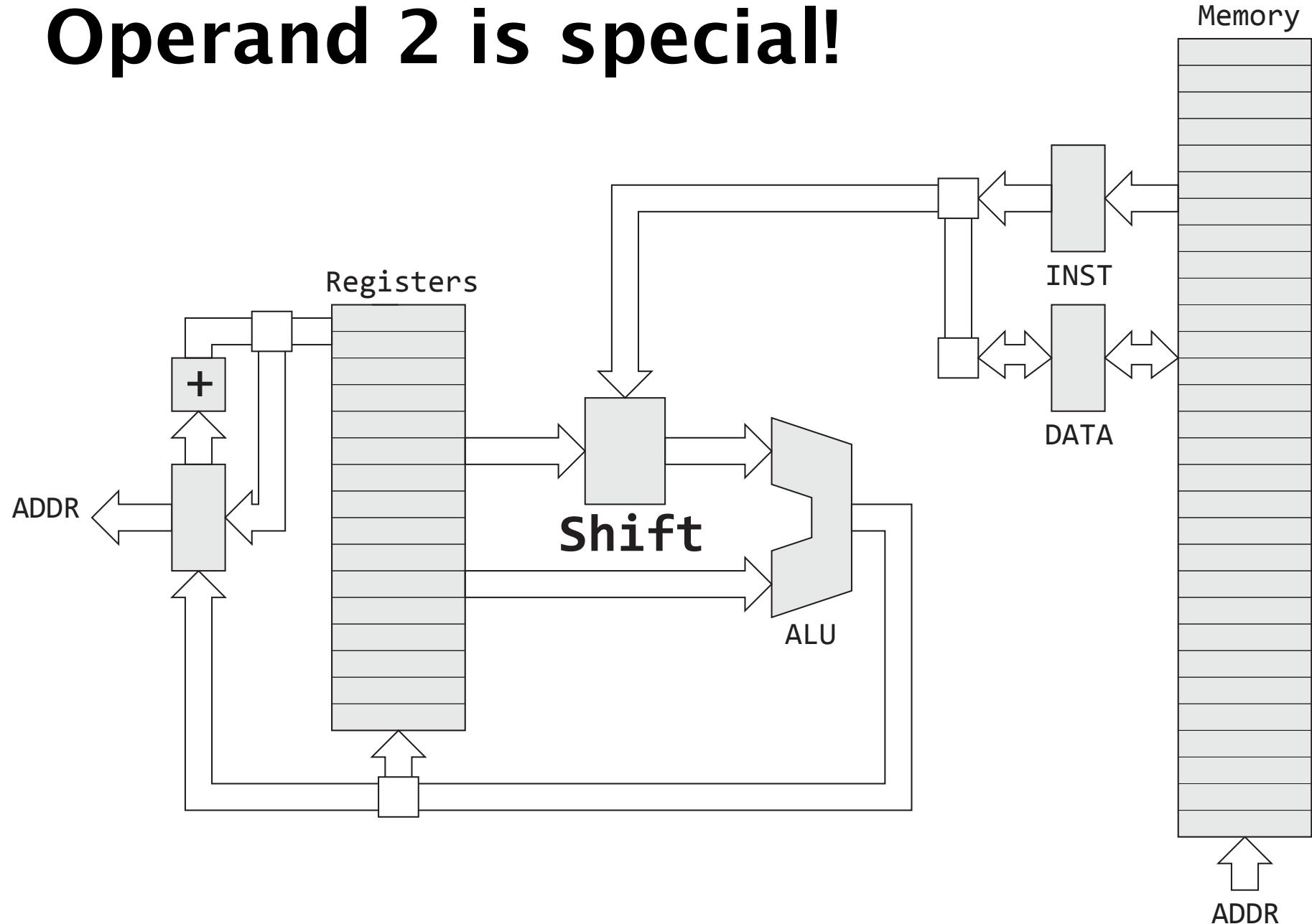
S bit on -> instruction will also set condition codes



`add r0, r1, r1`
`adds r0, r1, r1`
`add r0, r1, r1, lsl #1`



Operand 2 is special!



Condition Codes

- Z** set if result is 0, clear otherwise
- N** set if result is < 0, clear otherwise
- C** set if result generated carry
- V** set if result had arithmetic overflow

(More on carry and overflow later...)

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

Reads condition codes set by a previous instruction

(may be **cmp** or **tst** or other "s" instruction)

Branch taken if condition(s) satisfied

Challenge!

Write assembly program to count the number of "on" bits in a given number

```
mov r0, #val  
mov r1, #0  
  
// count bits in val in r0  
// put result in r1
```

VisUAL ARM Emulator

The screenshot shows the VisUAL ARM Emulator interface. The top bar includes buttons for New, Open, Save, Settings, Tools, Execute, Reset, Step Backwards, and Step Forwards. A status bar indicates "Emulation Complete" with 8 issues and 0 errors.

The assembly code in the editor is:

```
1 mov r0, #0x3a
2 mov r1, #0
3 loop
4     ands r2, r0, #1
5     addne r1, r1, #1
6     lsrs r0, r0, #1
7     bne loop
8
9
10
11
12
13
14
```

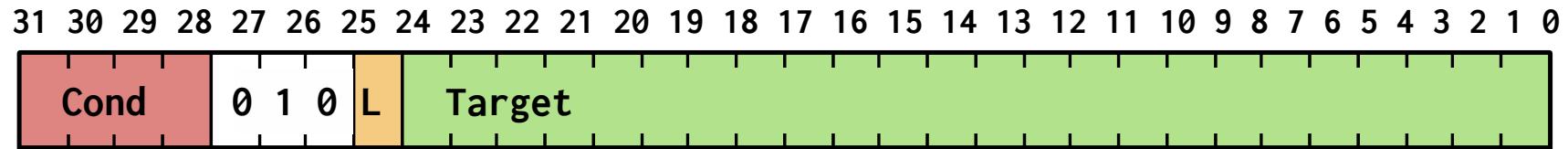
The registers table on the right shows the following values:

Register	Value	Dec	Bin	Hex
R0	0	Dec	Bin	Hex
R1	4	Dec	Bin	Hex
R2	1	Dec	Bin	Hex
R3	0	Dec	Bin	Hex
R4	0	Dec	Bin	Hex
R5	0	Dec	Bin	Hex
R6	0	Dec	Bin	Hex
R7	0	Dec	Bin	Hex
R8	0	Dec	Bin	Hex
R9	0	Dec	Bin	Hex
R10	0	Dec	Bin	Hex
R11	0	Dec	Bin	Hex
R12	0	Dec	Bin	Hex
R13	-16777216	Dec	Bin	Hex
LR	0	Dec	Bin	Hex
PC	32	Dec	Bin	Hex

At the bottom, there are counters for Clock Cycles (1) and Total (36), and a CSPR Status Bits (NZCV) field showing 0 1 1 0.

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

Branch instruction encoding



b (bal) branch always

1110 1010 tttt tttt tttt tttt tttt tttt

beq branch if zero flag 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?

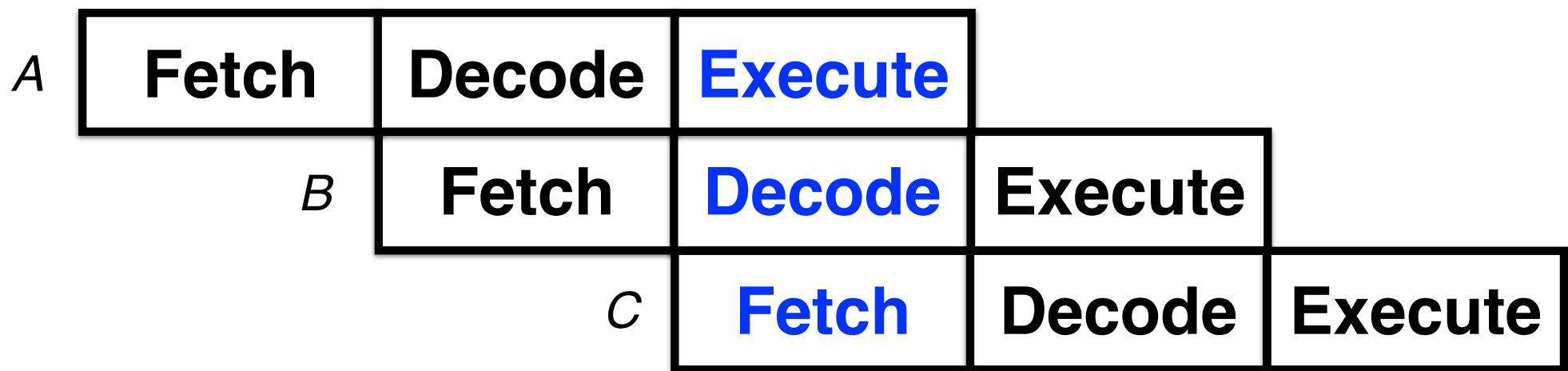
3 steps per instruction

Fetch	Decode	Execute
--------------	---------------	----------------

3 instructions takes 9 steps in sequence

...	Fetch	Decode	Execute	Fetch	...
-----	-------	--------	---------	-------	-----

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 (O). This is 2 instructions past A ($PC+8$)

```
18: e3a0283f mov      r2, #0x3fc000
1c: e2522001 subs     r2, r2, #1
20: 1afffffd bne      1c
```

```
// 1a          branch if not equal
// fffffd -3 (two's complement)
// offset is 8 + -3*4 = -4
```

PC-relative addressing used for data too!
(ldr/str)

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

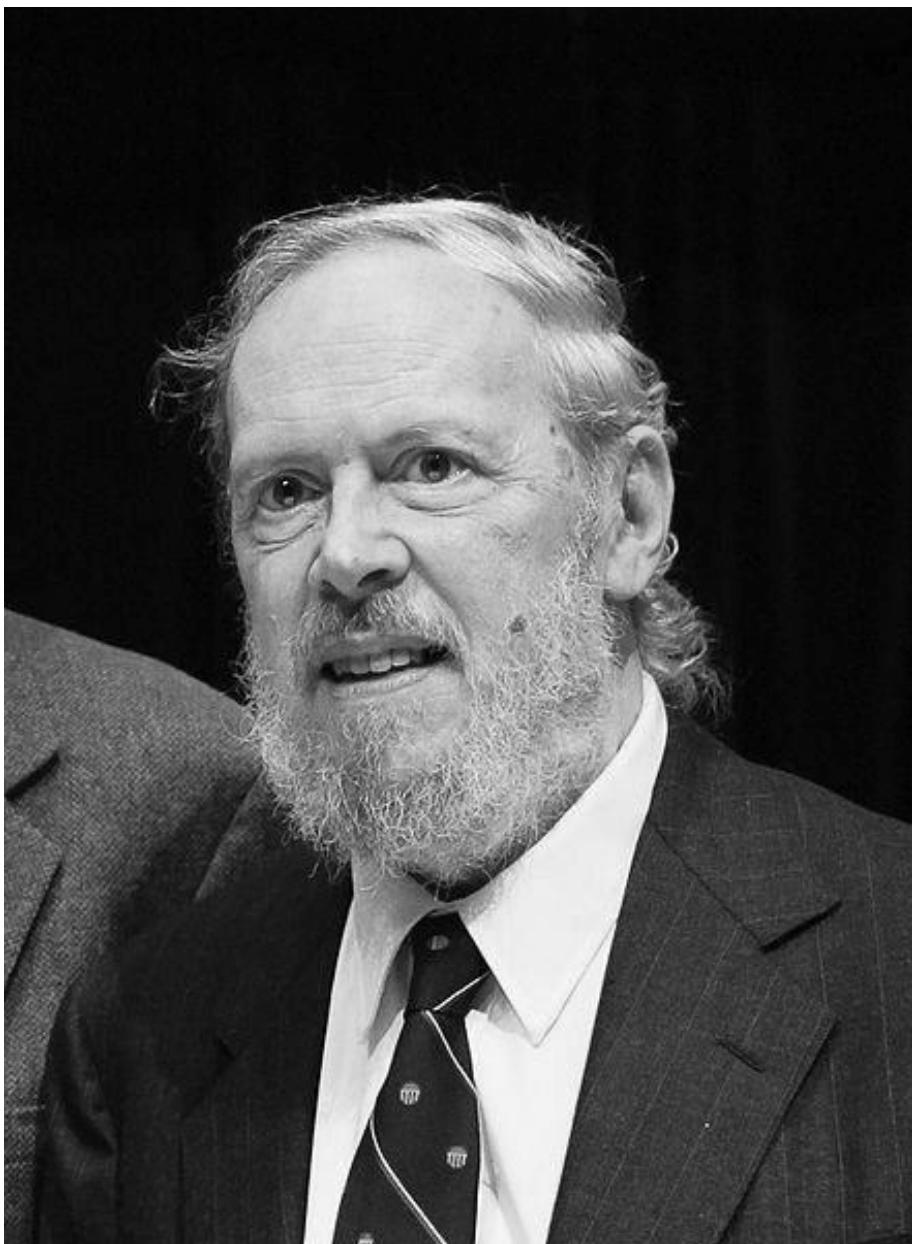
Set condition code (or not)

Orthogonality leads to composability

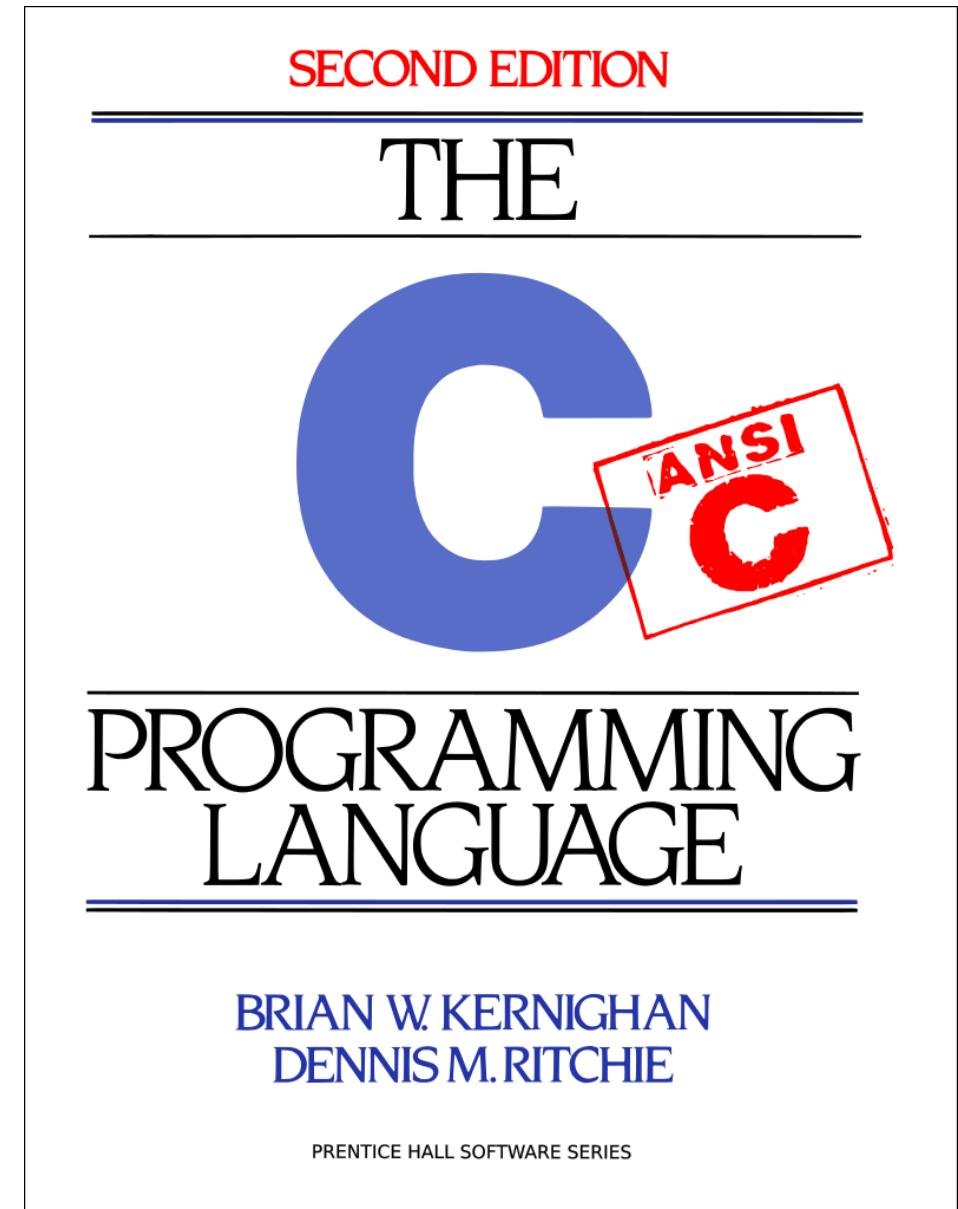


100 FEET BELOW
SEA LEVEL





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>

“BCPL, B, and C all fit firmly in the traditional procedural family (of languages) typified by Fortran and Algol 60. They 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.

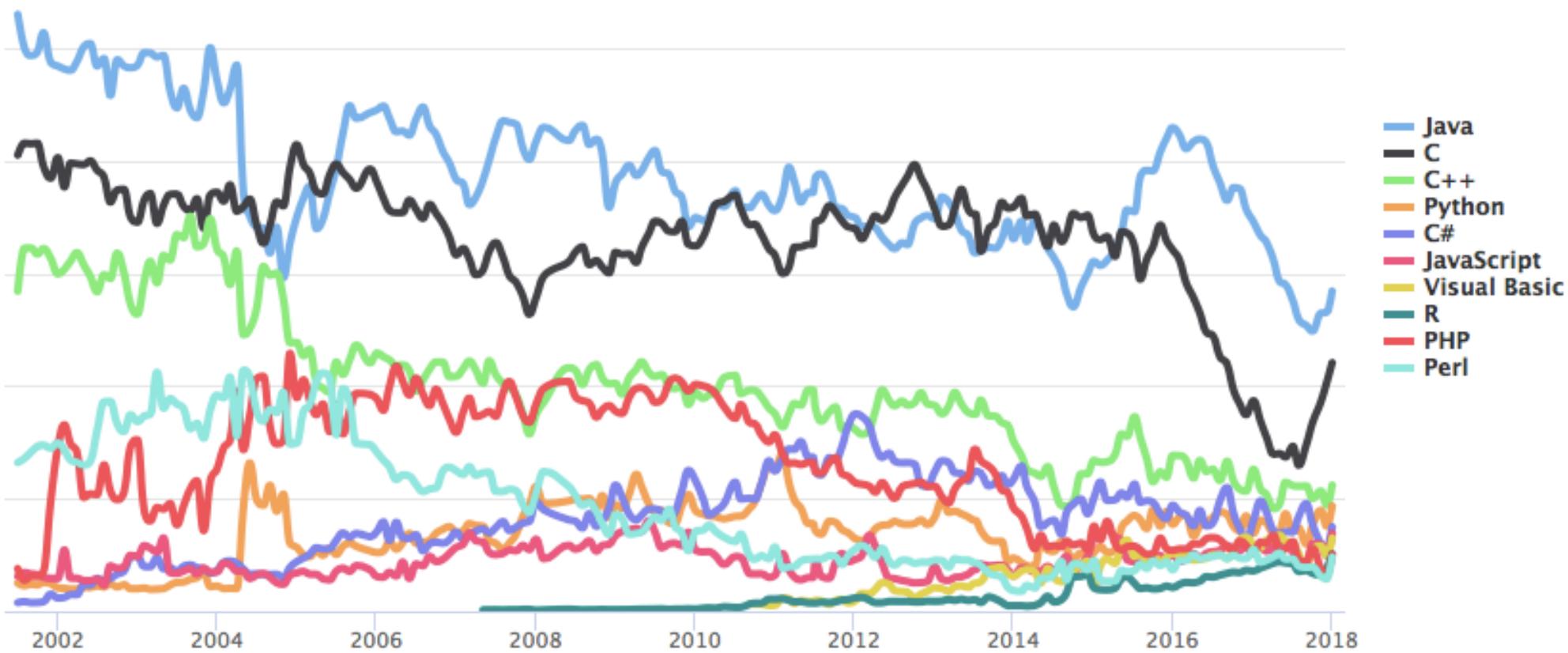
...

At the same time, their abstractions lie at a sufficiently high level that, with care, portability between machines can be achieved.”

- Dennis Ritchie

TIOBE Programming Community Index

Source: www.tiobe.com



Programming language popularity over time

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

The screenshot shows the Compiler Explorer interface. On the left, the C++ source code is displayed:

```
1 int global = 5;
2
3 void main(void)
4 {
5     global = global + 1;
6 }
```

The code editor has line numbers and syntax highlighting. The right side shows the generated assembly output for an ARM target:

```
11010 .LX0: .text // \s+ Intel Demangle
1 main:
2     ldr r2, .L2
3     ldr r3, [r2]
4     add r3, r3, #1
5     str r3, [r2]
6     bx lr
```

The assembly code is color-coded by instruction type. The tabs at the bottom allow switching between different assembly formats: 11010, .LX0, .text, //, \s+, Intel, and Demangle.

<https://gcc.godbolt.org>

```
.equ DELAY, 0x3F0000
```

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

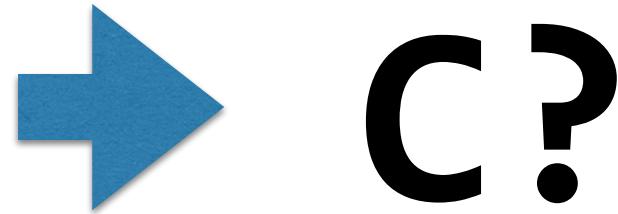
```
loop:
```

```
    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
```



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* ?