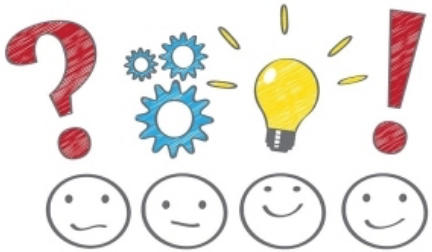
Admin

First lab and assign

We're off and running!

Check in



Today: From Assembly to C (and back again)

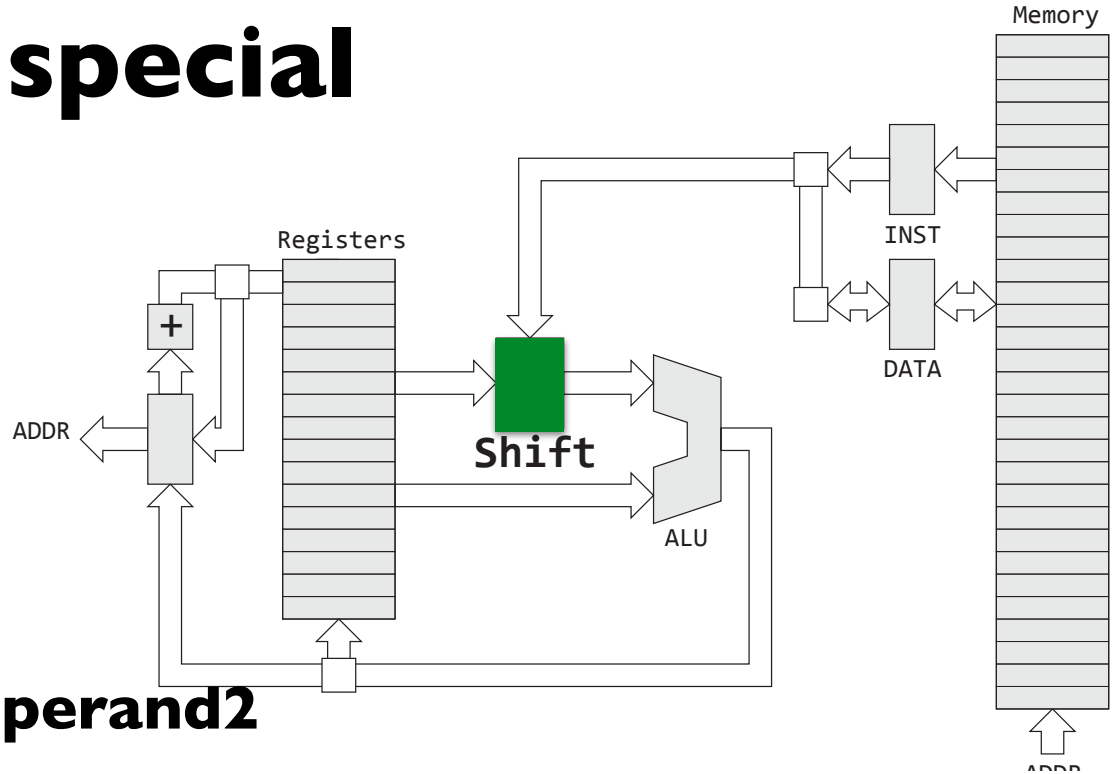
ARM condition codes, branch instructions

C language as "high-level" assembly

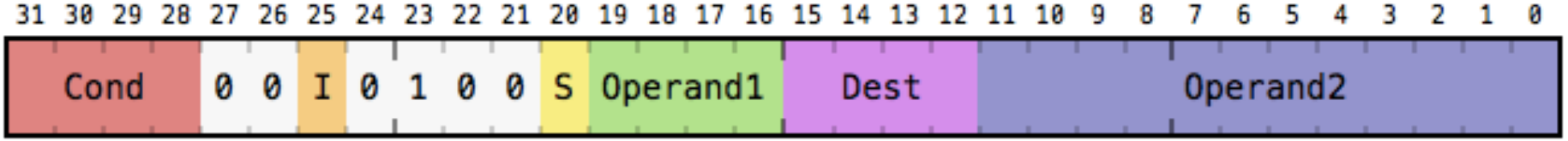
What does a compiler do?

Makefiles

Operand 2 is special

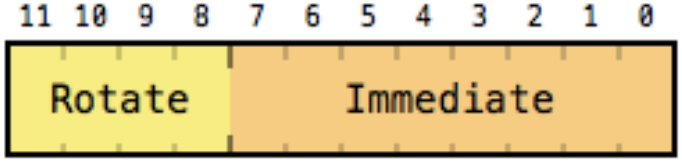


Dest = Operand1 op Operand2

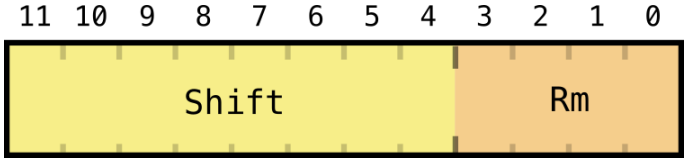



```
add r0, r1, #0x1f000
sub r0, r1, #6
rsb r0, r1, #6
add r0, r1, r2, lsl #3
mov r1, r2, ror #7
```

I=1



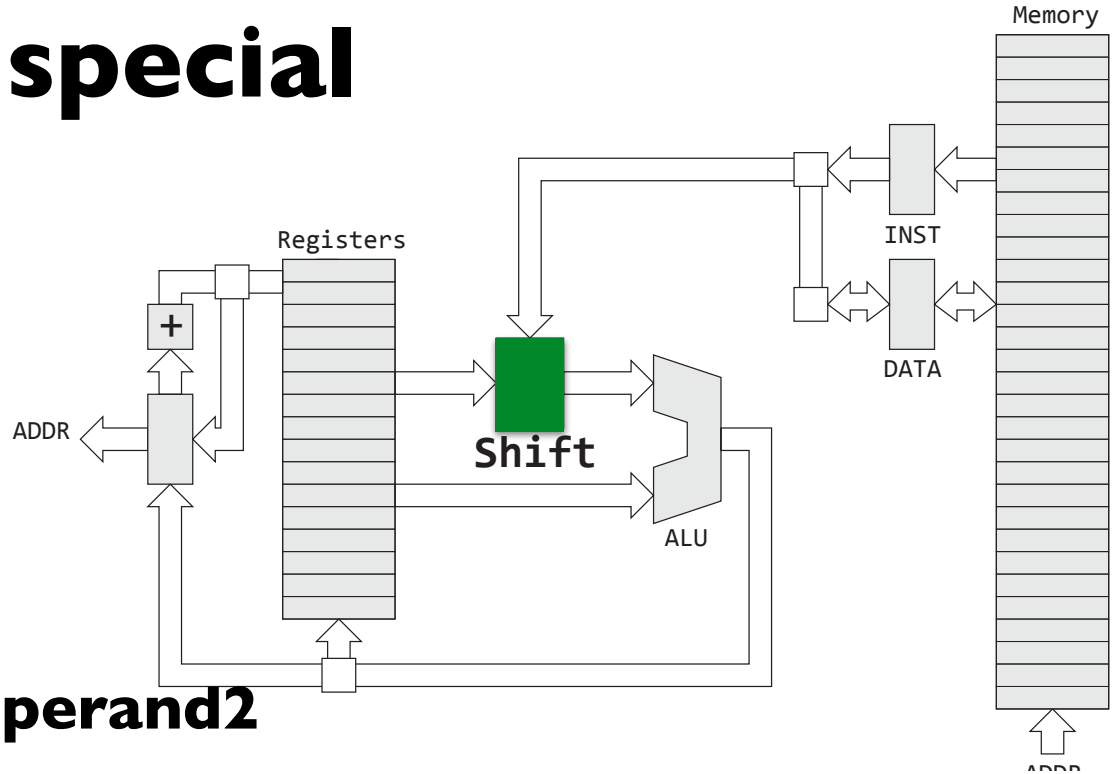
I=0



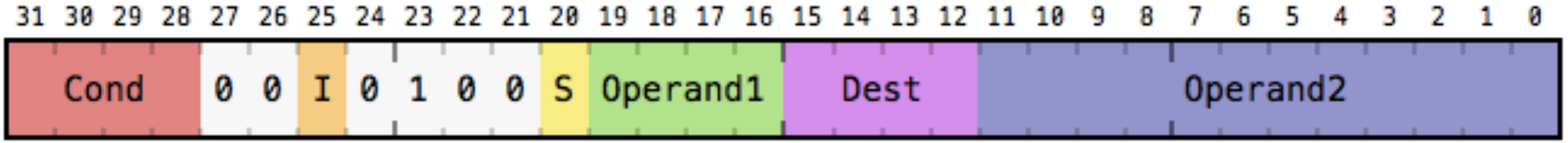
```
///  SET0 = 0x2020001c
mov r0, #0x20        // r0 = 0x00000020
lsl r1, r0, #24      // r1 = 0x20000000
lsl r2, r0, #16      // r2 = 0x00200000
orr r0, r1, r2       // r0 = 0x20200000
orr r0, r0, #0x1c    // r0 = 0x2020001c

//  SET0 = 0x2020001c
mov r0, #0x20000000 // 0x20>>>8
orr r0, #0x00200000 // 0x20>>>16
orr r0, #0x0000001c // 0x1c>>>0
```

Rotation of small constants makes code 40% shorter (and 40% faster)

Operand 2 is special

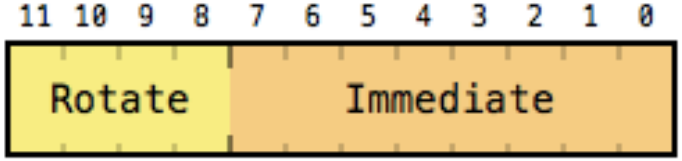


Dest = Operand1 op Operand2

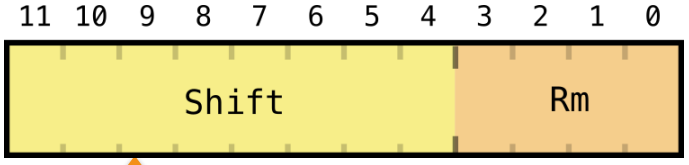



```
add r0, r1, #0x1f000
sub r0, r1, #6
rsb r0, r1, #6
add r0, r1, r2, lsl #3
mov r1, r2, ror #7
```

I=1

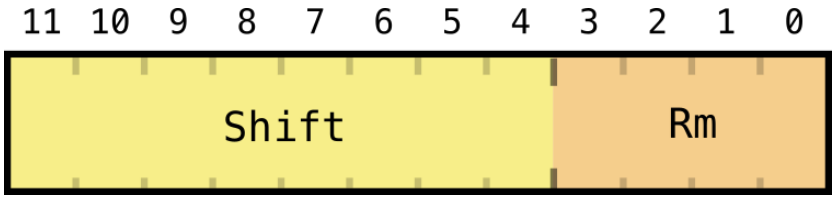


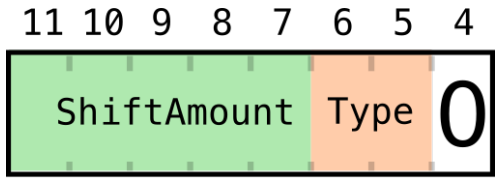
I=0



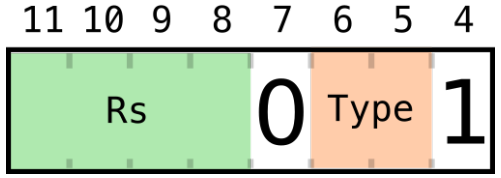
lsl, lsr, asr, ror

Shift Operand Details





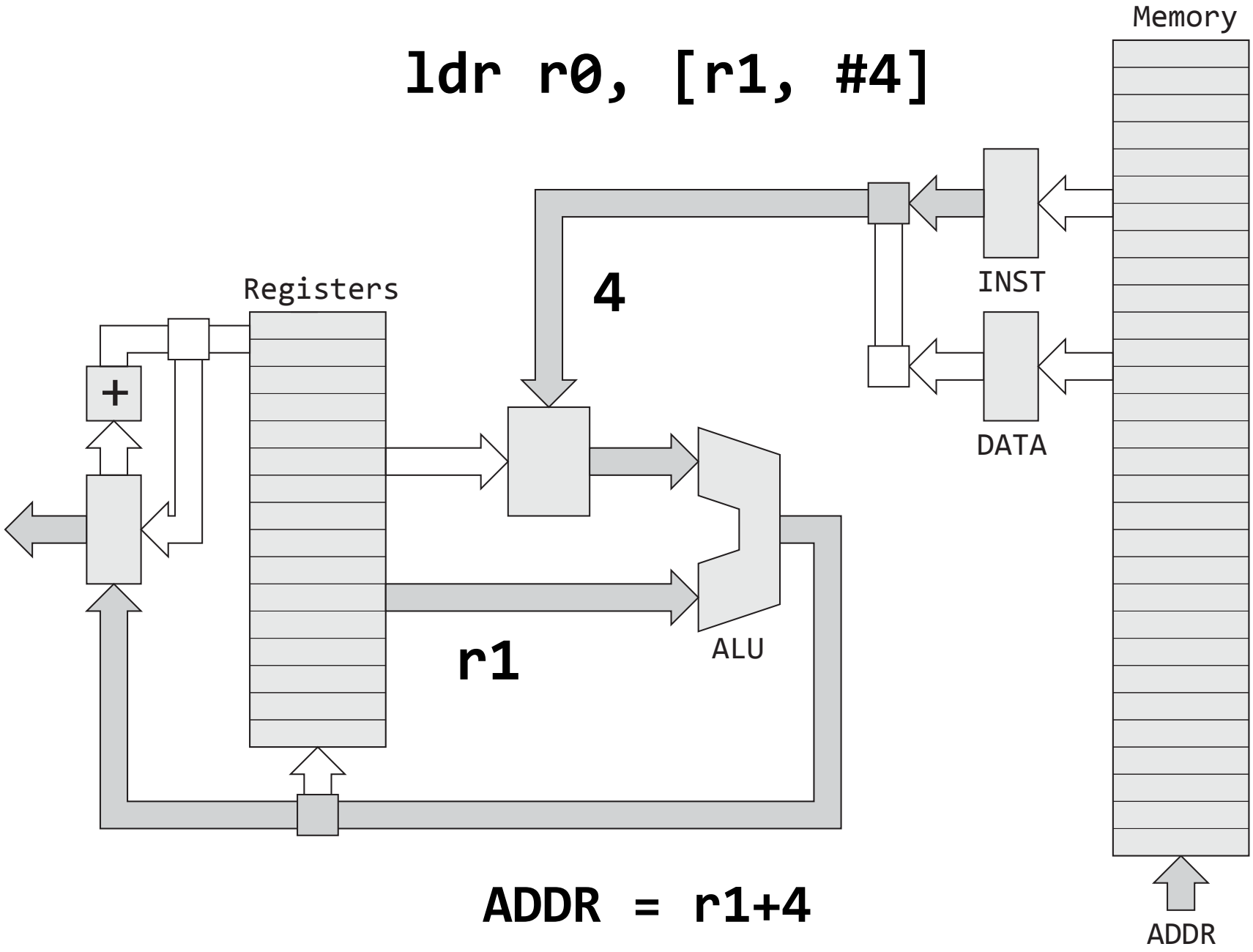
Shift Rm by ShiftAmount



Shift Rm by Rs

00: lsl (logical shift left)
01: lsr (logical shift right)
10: asr (arithmetic shift right)
11: ror (rotate right)

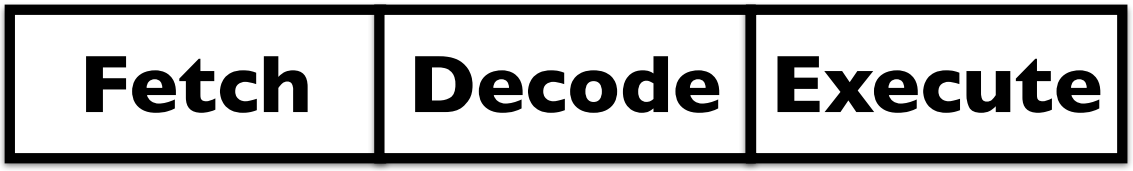
Load from Memory to Register (LDR)



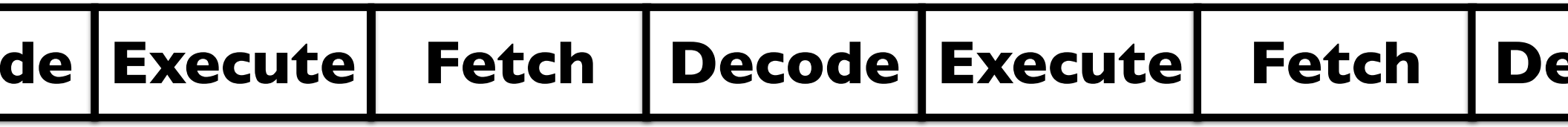
```
// configure GPIO 20 for output
ldr r0, =0x20200008
mov r1, #1
str r1, [r0]

// set bit 20
ldr r0, =0x2020001C
mov r1, #0x00100000
str r1, [r0]

loop: b loop
```

```
// configure GPIO 20 for output
ldr r0, [pc + 20]
mov r1, #1
str r1, [r0]

// set bit 20
ldr r0, [pc  + 12]
mov r1, #0x00100000
str r1, [r0]

loop: b loop

.word 0x20200008
.word 0x2020001C
```

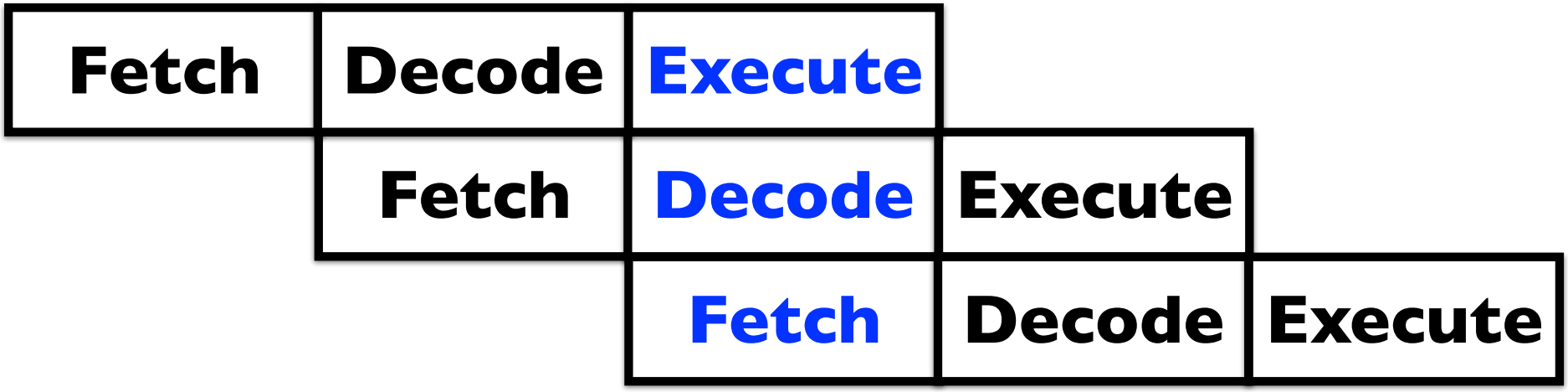
3 steps to run an instruction



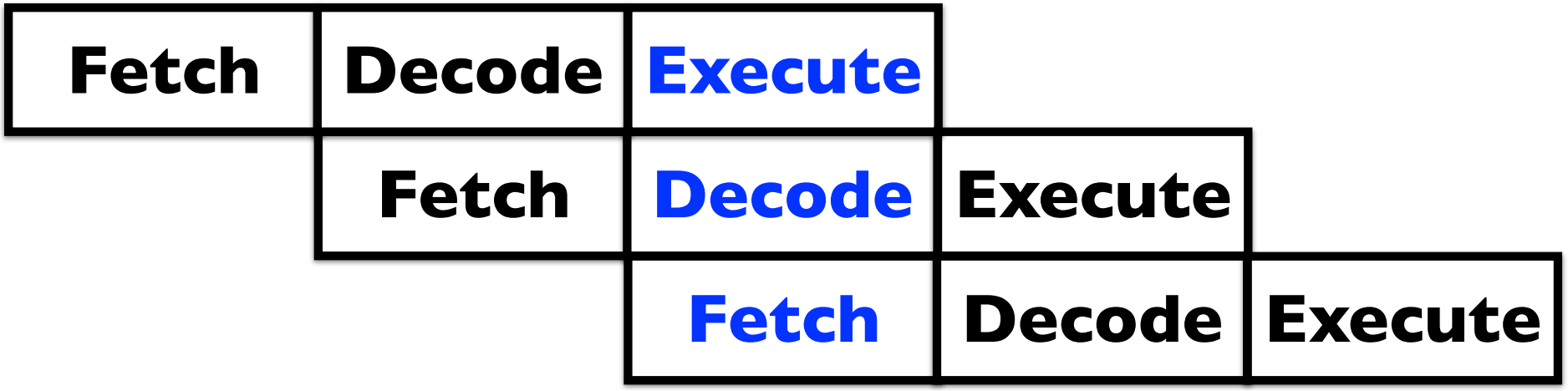
3 instructions takes 9 steps



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



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



PC value in the executing instruction is equal to the pc value of the instruction being fetched - which is 2 instructions ahead (PC+8)

Changing Control Flow

Loops, branches, and condition codes

Control flow

Instructions stored in memory contiguously

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

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

branch instructions change what instruction is fetched, decoded, and executed next has effect of **pc = target**

```
b target
```

Above branch is unconditional (always taken)

Branches can also be 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…)

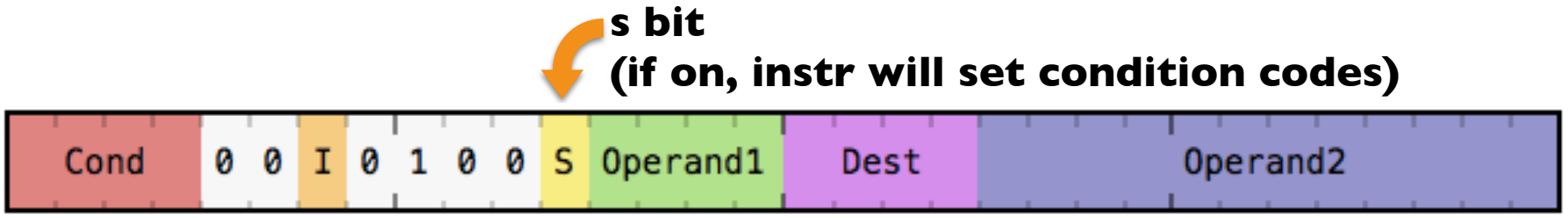
Which instructions set/clear codes?

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

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

Any data processing instruction suffixed with **s:**

adds movs orrs lsrs ...



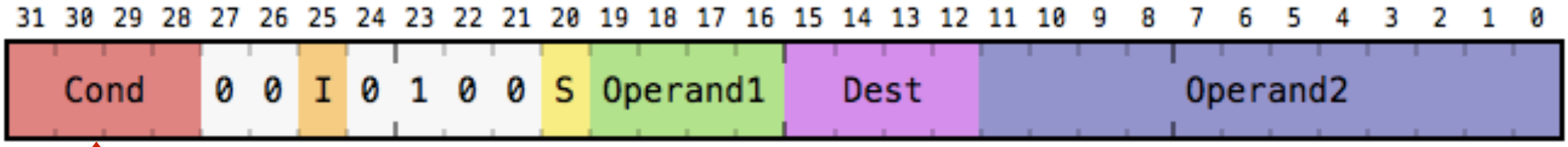
Branch instructions

b target
bne target
bmi target
bge target
...

branch reads condition codes, as set by a previous instruction

If specific condition satisfied, branch is taken, **pc = target**
otherwise falls through, **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****?*

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

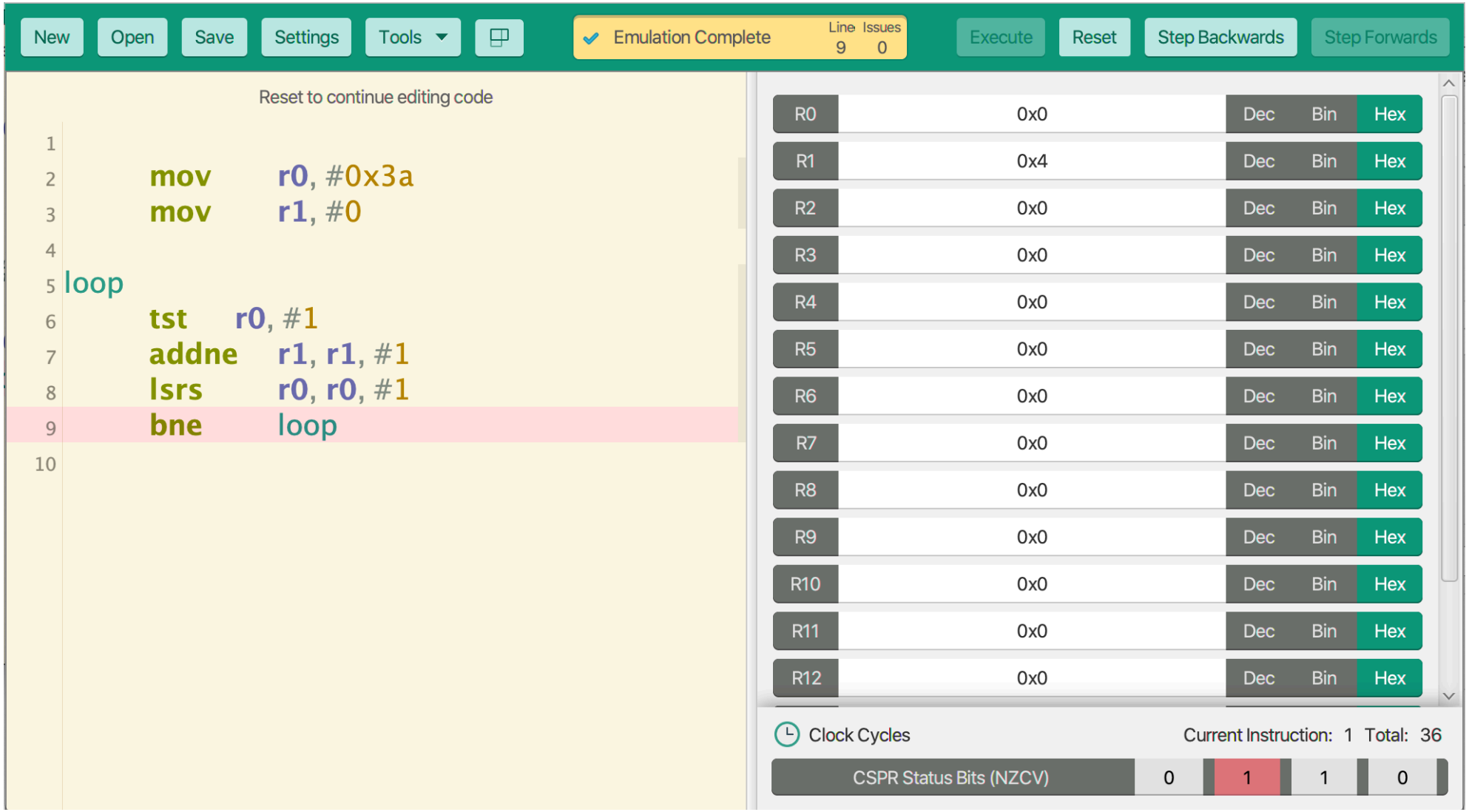
Challenge for you all:

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

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

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

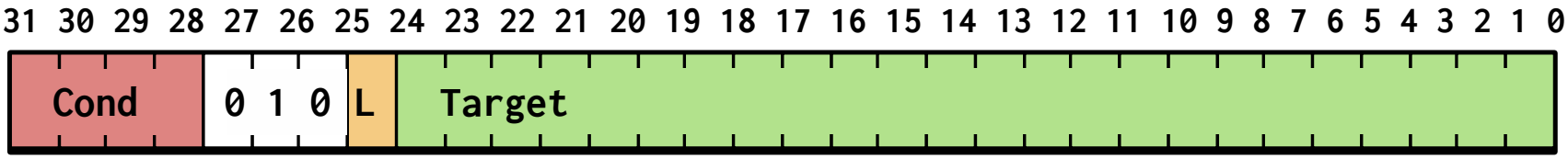
https://salmanarif.bitbucket.io/visual/

VisUAL ARM Emulator



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

Branch instruction encoding



b (bal) branch always

1110 1010 tttt tttt tttt tttt tttt tttt

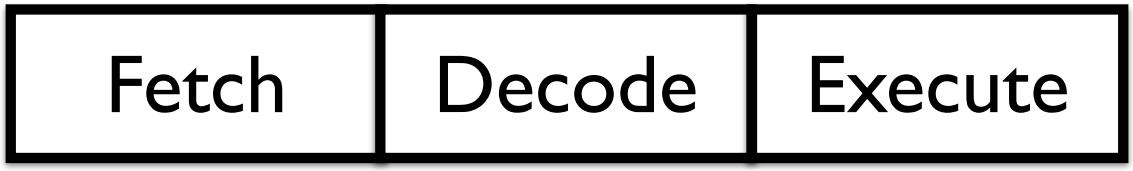
beq branch if zero CC set

0000 1010 tttt tttt tttt tttt tttt tttt

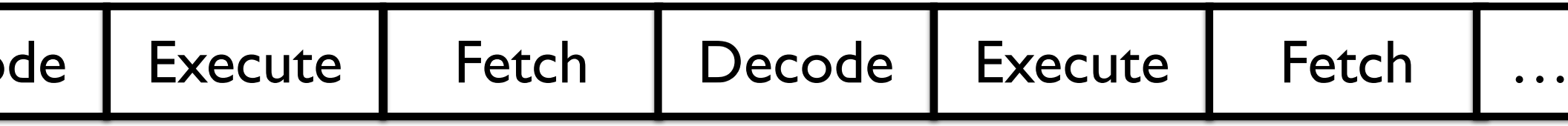
branch target is PC-relative offset
green bits encode offset, counted in 4-byte words

Q: How far can this reach?

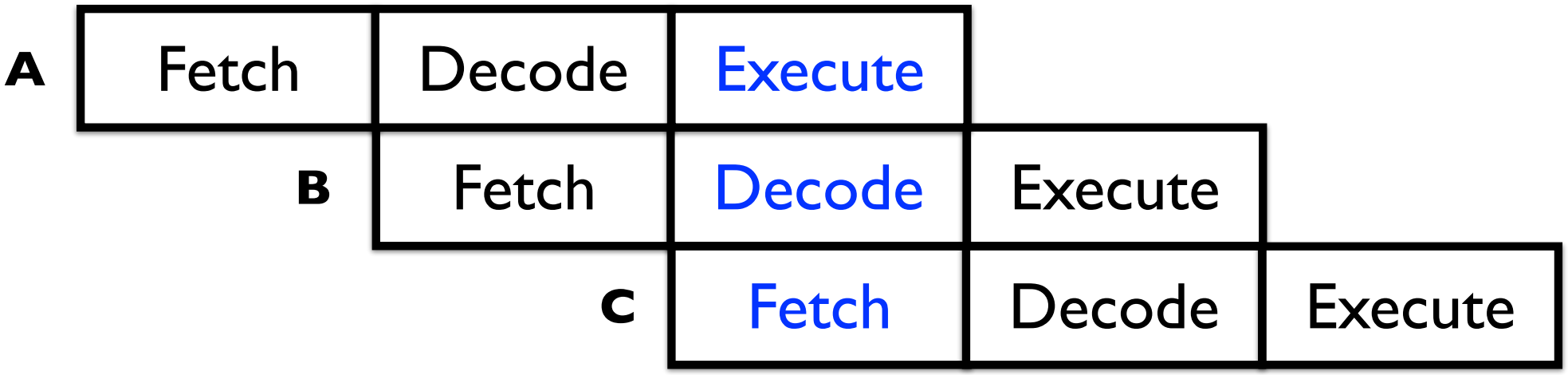
3 steps per instruction



3 instructions takes 9 steps in sequence

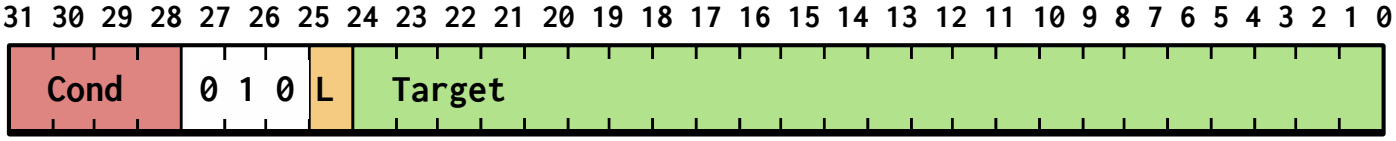


To speed things up, steps are overlapped ("pipelined")



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

Why assembly?

What you see is what you get

No surprises

Precise control, timing

Unfettered access to hardware

Why C?

More concise

Easier to read

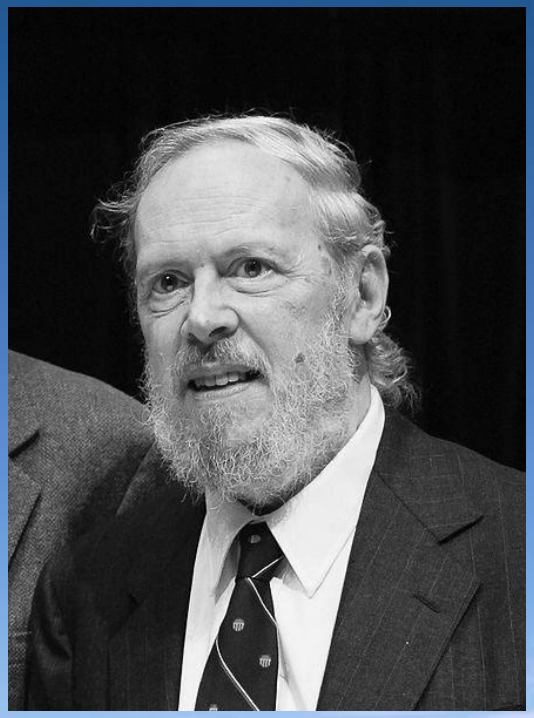
Can name variables and structures

Type-checking

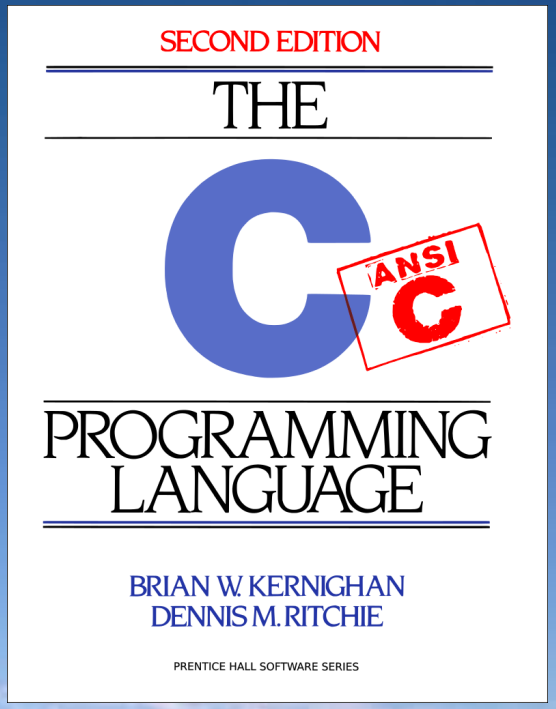
More portable (ARMv6, ARMv7, ARMv8...)

Functions!!

Real question is not whether to use assembly, but ***when****...*



Dennis Ritchie



C is the language of choice for systems programmers

Ken Thompson built UNIX using C

This is not coincidence!

C features closely model the ISA: data types, arithmetic/logical operators, control flow, access to memory, ... all provided in form of portable abstractions

"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*

The C Programming Language

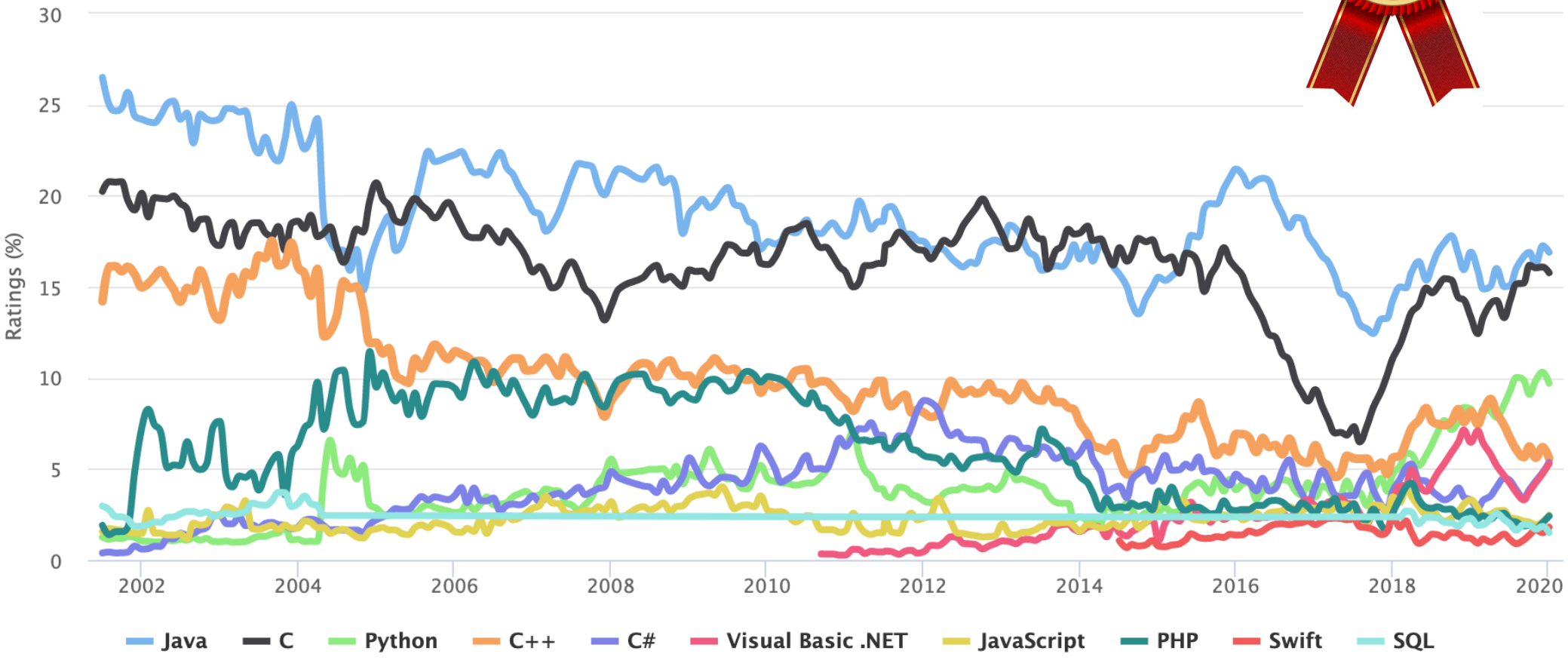
"C is quirky, flawed, and an enormous success"
— *Dennis Ritchie*

"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*

January 2020 headline:

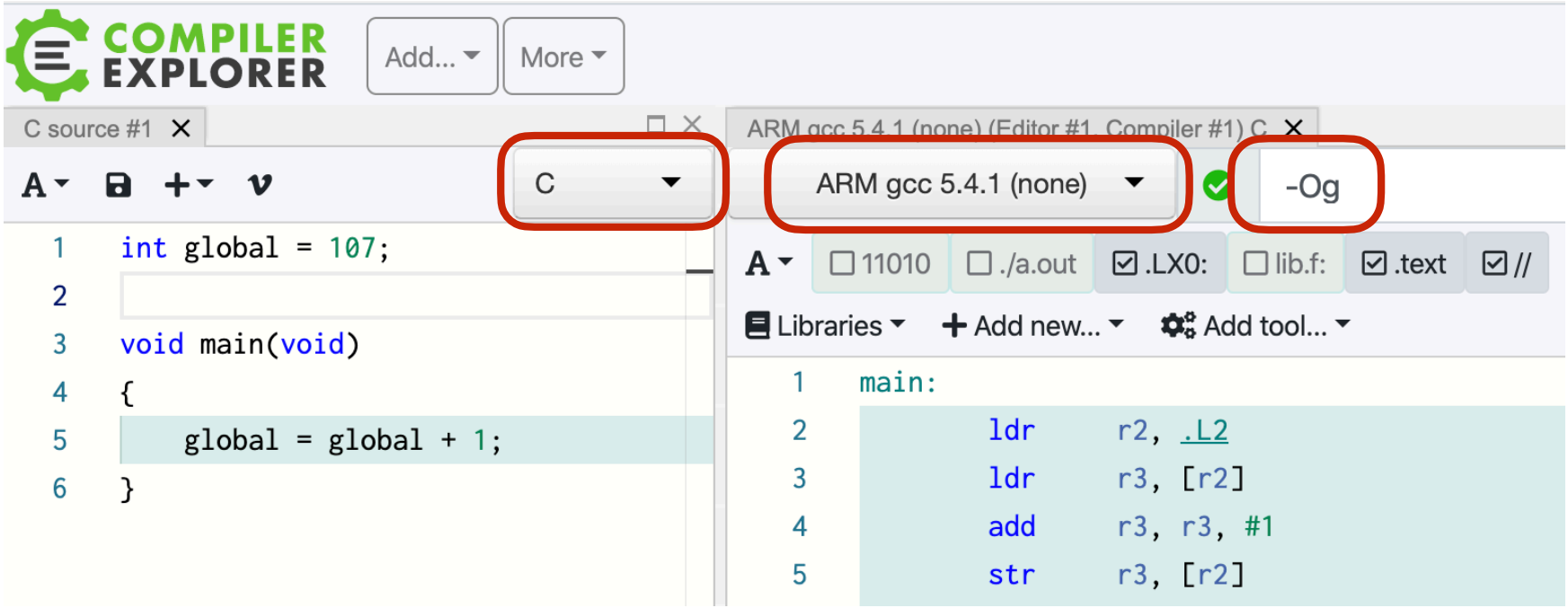
Programming Language of the Year 2019....



Programming language popularity over time

Compiler Explorer

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



https://godbolt.org

Configure settings to follow along:
C
ARM gcc 5.4.1 (none)
-Og

Why C?

Higher-level abstractions, structured programming

- Named variables, constants
- Arithmetic/logical operators
- Control flow

Portable

- Not tied to particular ISA or architecture

Low-level enough to get to machine when needed

- Bitwise operations
- Direct access to memory
- Embedded assembly, too!

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

Make

Build/compile your code using **make**

A **Makefile** describes how to build a piece of software

Rules, dependencies, and recipes

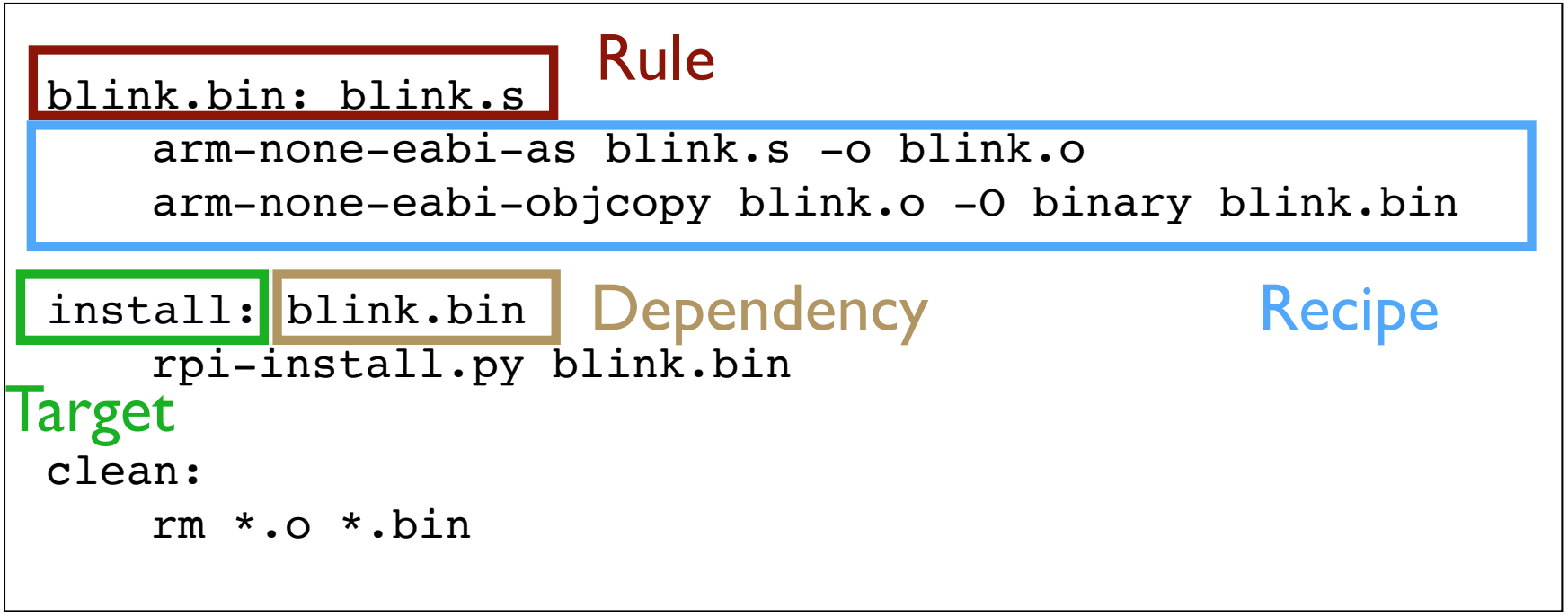
```
mine.bin: mine.s
    arm-none-eabi-as mine.s -o mine.o
    arm-none-eabi-objcopy mine.o -O binary mine.bin

install: mine.bin
    rpi-install.py mine.bin

clean:
    rm *.o *.bin
```

Make

Build/compile your code using **make**

A **Makefile** describes how to build a piece of software

Rules, targets, dependencies, and recipes



Make

Build/compile your code using **make**

A **Makefile** describes how to build a piece of software

Rules, targets, dependencies, and recipes

```
blink.bin: blink.s
    arm-none-eabi-as blink.s -o blink.o
    arm-none-eabi-objcopy blink.o -O binary blink.bin
```

Here's the rule to make build.bin

It depends on build.s; if that changes, run this recipe again

Here are the steps (recipe) to produce build.bin

Make

Build/compile your code using **make**

A **Makefile** describes how to build a piece of software

Rules, targets, dependencies, and recipes

```
blink.bin: blink.s
    arm-none-eabi-as blink.s -o blink.o
    arm-none-eabi-objcopy blink.o -O binary blink.bin
```

Here's the rule to make build.bin

It depends on build.s; if that changes, run this recipe again

Writing out and recipes explicitly like this is onerous, so make has all kinds of ways to match patterns, define variables, etc.

Make pattern rules

```
NAME = myprogram

CFLAGS = -Og -g -Wall -std=c99 -ffreestanding
LDFLAGS = -nostdlib -e main

all : $(NAME).bin

%.bin: %.elf
	arm-none-eabi-objcopy $< -O binary $@

%.elf: %.o
	arm-none-eabi-gcc $(LDFLAGS) $< -o $@

%.o: %.c
	arm-none-eabi-gcc $(CFLAGS) -c $< -o $@

%.o: %.s
	arm-none-eabi-as $< -o $@
```

Bare-metal vs. Hosted

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 supply our own replacements where needed

Build settings for bare-metal

Compile freestanding

CFLAGS = -ffreestanding

Link excludes standard library and 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…