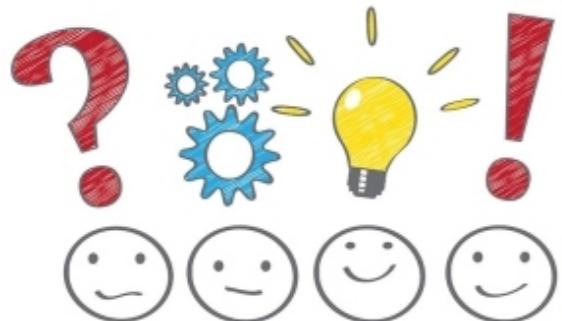


# Admin

*First lab and assign  
We're off and running!*

*Check in*



## Goals for today

More ARM: condition codes, branches

C language as “high-level” assembly

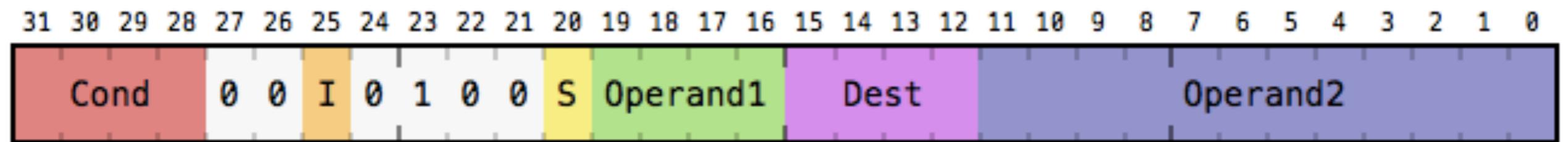
How to compile/build from C source, Makefile

Relationship of C to asm, study translations

```

# pattern for data processing instructions
#     dst = operand1 op operand2 | imm
#
# I - immediate
# S - set condition code

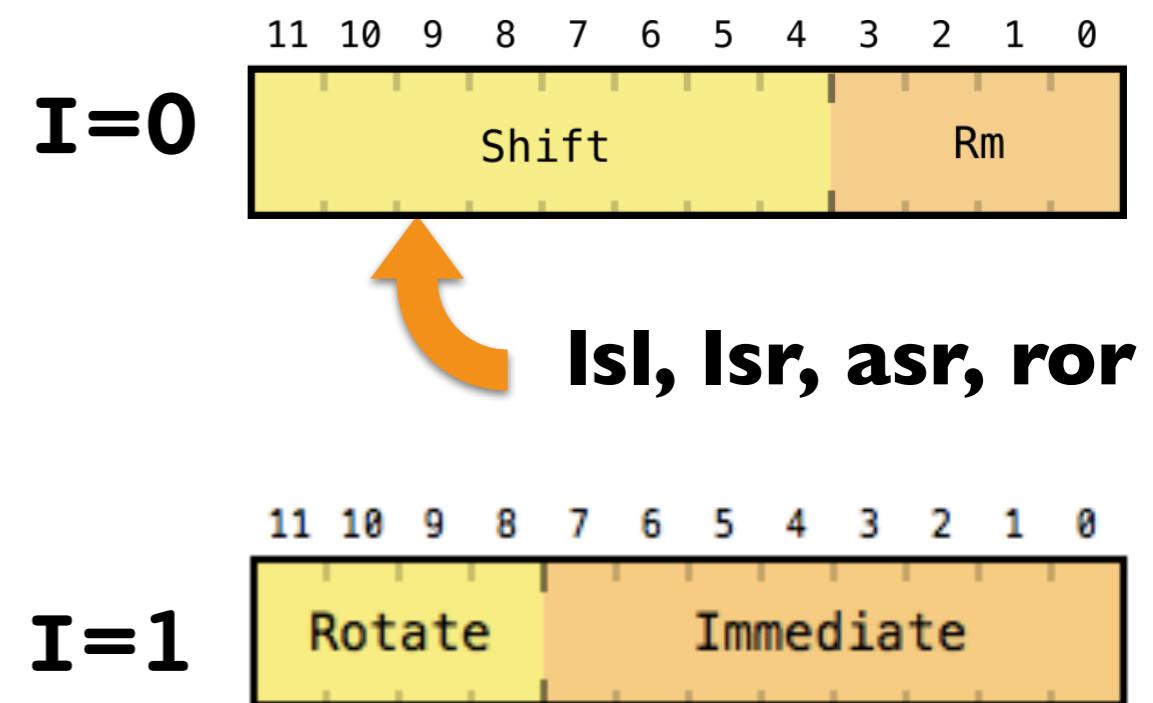
```



```

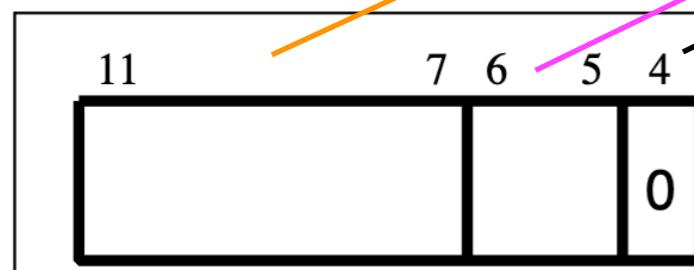
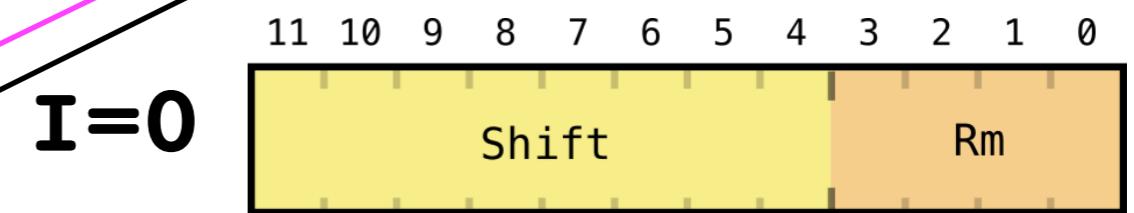
# examples
sub r0, r1, #107
orr r0, r0, r1
add r0, r0, r2, lsl #1

```



add r0, r0, r2, lsl #1

1110 0000 1000 0000 0000 0000 1000 0010

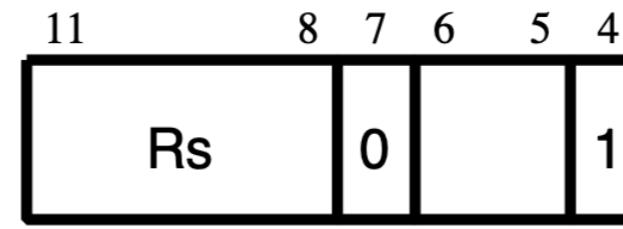


#### Shift type

- 00 = logical left
- 01 = logical right
- 10 = arithmetic right
- 11 = rotate right

#### Shift amount

5 bit unsigned integer



#### Shift type

- 00 = logical left
- 01 = logical right
- 10 = arithmetic right
- 11 = rotate right

#### Shift register

Shift amount specified in bottom byte of Rs

Figure 4-5: ARM shift operations

# 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"

# VisUAL ARM Emulator

The screenshot shows the VisUAL ARM Emulator interface. The assembly code in the editor window is:

```
1      mov    r0, #0
2      mov    r1, #0
3 anna  cmp    r1, #20
4      bgt   done
5      add    r0, r0, r1
6      add    r1, r1, #1
7      b     anna
8 done
9      end
```

The register values are:

Register	Value	Dec	Bin	Hex
R0	0x0	Dec	Bin	Hex
R1	0x0	Dec	Bin	Hex
R2	0x0	Dec	Bin	Hex
R3	0x0	Dec	Bin	Hex
R4	0x0	Dec	Bin	Hex
R5	0x0	Dec	Bin	Hex
R6	0x0	Dec	Bin	Hex
R7	0x0	Dec	Bin	Hex
R8	0x0	Dec	Bin	Hex
R9	0x0	Dec	Bin	Hex
R10	0x0	Dec	Bin	Hex
R11	0x0	Dec	Bin	Hex
R12	0x0	Dec	Bin	Hex
R13	0xFF000000	Dec	Bin	Hex
LR	0x0	Dec	Bin	Hex
PC	0x0	Dec	Bin	Hex

At the bottom, there are status indicators for Clock Cycles (0), Current Instruction (0 Total: 0), and CSPR Status Bits (NZCV) with values 0 0 0 0.

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

# 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** ...

 **s bit**  
**(if on, instr will set condition codes)**



<b>Code</b>	<b>Suffix</b>	<b>Flags</b>	<b>Meaning</b>
0000	EQ	Z set	equal
0001	NE	Z clear	not equal
0010	CS	C set	unsigned higher or same
0011	CC	C clear	unsigned lower
0100	MI	N set	negative
0101	PL	N clear	positive or zero
0110	VS	V set	overflow
0111	VC	V clear	no overflow
1000	HI	C set and Z clear	unsigned higher
1001	LS	C clear or Z set	unsigned lower or same
1010	GE	N equals V	greater or equal
1011	LT	N not equal to V	less than
1100	GT	Z clear AND (N equals V)	greater than
1101	LE	Z set OR (N not equal to V)	less than or equal
1110	AL	(ignored)	always

# Branch instructions

**b target**

**bne target**

**bmi target**

**bge target**

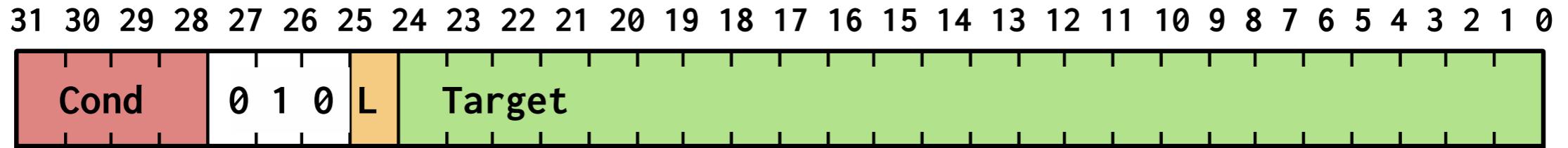
...

**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**

# Branch instruction encoding



**b (bal) branch always**

**1110 1010 tttt tttt tttt tttt tttt tttt**

**beq branch if zero CC set**

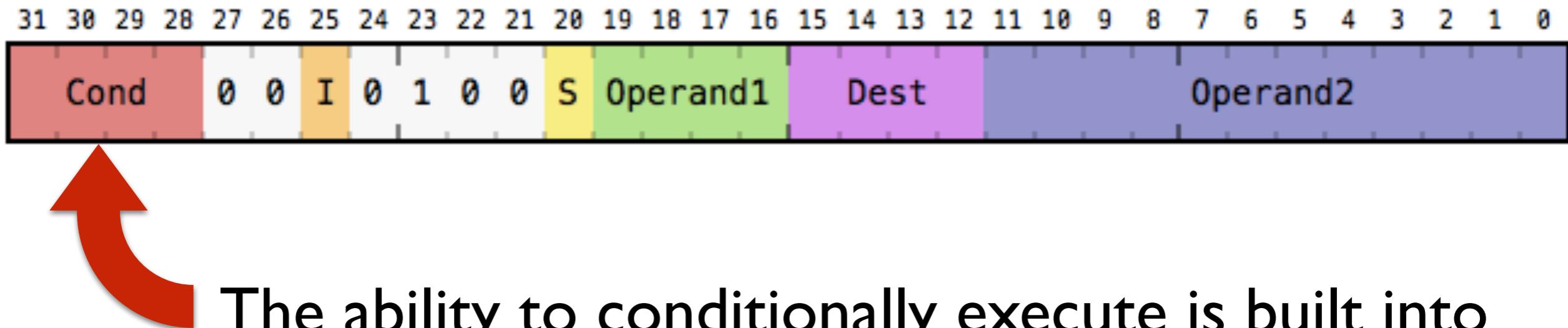
**0000 1010 tttt tttt tttt tttt tttt tttt**

branch target is PC-relative offset

green bits encode offset, counted in 4-byte words

*Q: How far can this reach?*

# 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**?*

# An assembly program to count the "on" bits in a given numeric value

The screenshot shows the VisUAL assembly editor interface. The assembly code in the left pane is:

```
1    mov    r0, #0x3a
2    mov    r1, #0
3
4 anna
5    tst    r0, #1
6    addne r1, r1, #1
7    lsrs   r0, r0, #1
8    bne    anna
9
10   end
```

The right pane displays the register state:

Register	Value	Dec	Bin	Hex
R0	0x0	Dec	Bin	Hex
R1	0x0	Dec	Bin	Hex
R2	0x0	Dec	Bin	Hex
R3	0x0	Dec	Bin	Hex
R4	0x0	Dec	Bin	Hex
R5	0x0	Dec	Bin	Hex
R6	0x0	Dec	Bin	Hex
R7	0x0	Dec	Bin	Hex
R8	0x0	Dec	Bin	Hex
R9	0x0	Dec	Bin	Hex
R10	0x0	Dec	Bin	Hex
R11	0x0	Dec	Bin	Hex
R12	0x0	Dec	Bin	Hex
R13	0xFF000000	Dec	Bin	Hex
LR	0x0	Dec	Bin	Hex
PC	0x0	Dec	Bin	Hex

At the bottom, there are status indicators for Clock Cycles (0), Current Instruction (0), Total (0), and CSPR Status Bits (NZCV) with values 0, 0, 0, 0.

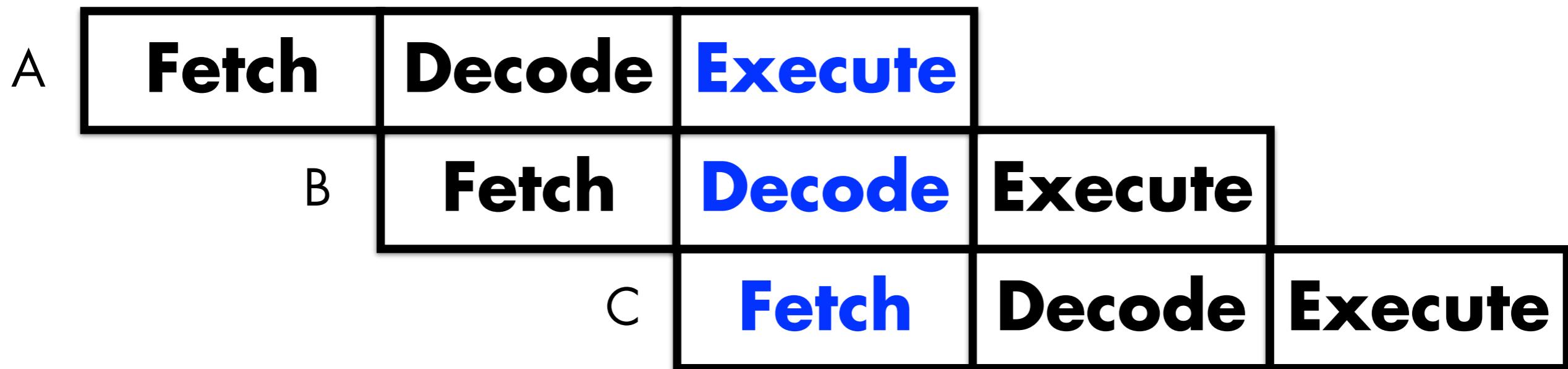
# **3 steps per instruction**

<b>Fetch</b>	<b>Decode</b>	<b>Execute</b>
--------------	---------------	----------------

## **3 instructions takes 9 steps in sequence**

Decode	Execute	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 (C). This is 2 instructions past A (PC+8)**

**<delay>:**

```
24: e3a039ff  mov      r3, #0x3fc000
28: e2533001  subs     r3, r3, #1
2c: 1affffffc bne     24 <delay>
30:
34:
```

```
// 1a = branch if not equal
// ffffffc = -4 (two's complement)
// pc = (pc + 8) -4*4 (word size)
```

**PC-relative addressing used for data  
too (more on that next lecture...)**

# **ISA Design is an Art Form!**

**Commonalities across operations**

**Register vs. immediate operands**

**All registers the same\*\***

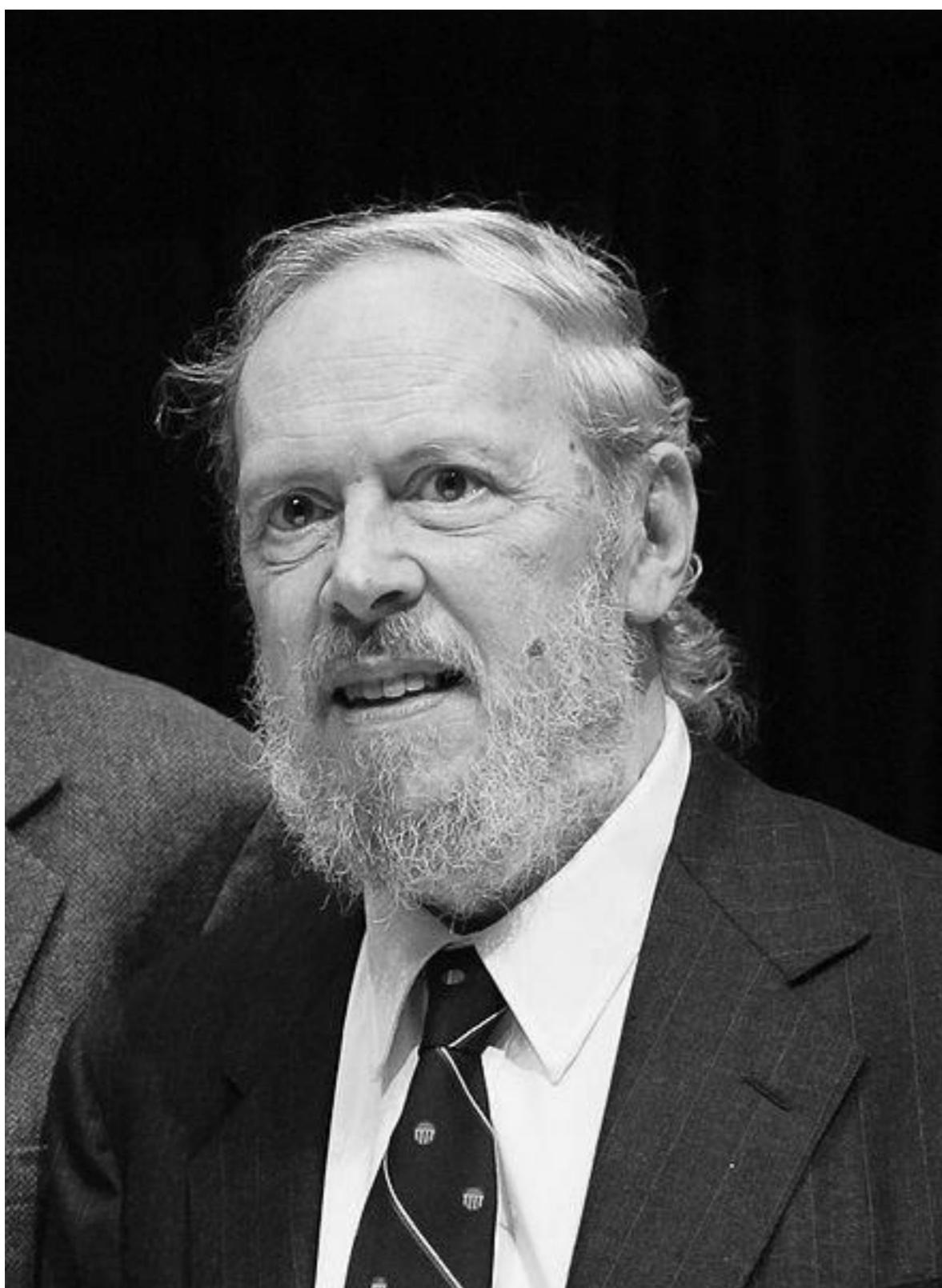
**Predicated/conditional execution**

**Set condition code (or not)**

**Orthogonality leads to composability**



100 FEET BELOW  
SEA LEVEL



**Dennis Ritchie**

SECOND EDITION

---

THE

---



---

PROGRAMMING  
LANGUAGE

---

BRIAN W. KERNIGHAN  
DENNIS M. RITCHIE

PRENTICE HALL SOFTWARE SERIES

# 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

VS.

```
void main(void)
{
    // configure GPIO 20 as output
    *(unsigned int *)0x20200008 = 1;

    // set GPIO 20 high
    *(unsigned int *)0x2020001C = 1 << 20;
}
```

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

// set GPIO 20 high
ldr r0, =0x2020001C
mov r1, #(1<<20)
str r1, [r0]
```

# 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
```

see <http://cs107e.github.io/guides/make/>

# 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

Rule

    arm-none-eabi-as blink.s -o blink.o

    arm-none-eabi-objcopy blink.o -O binary blink.bin

install: blink.bin

Dependency

    rpi-install.py blink.bin

Recipe

Target

clean:

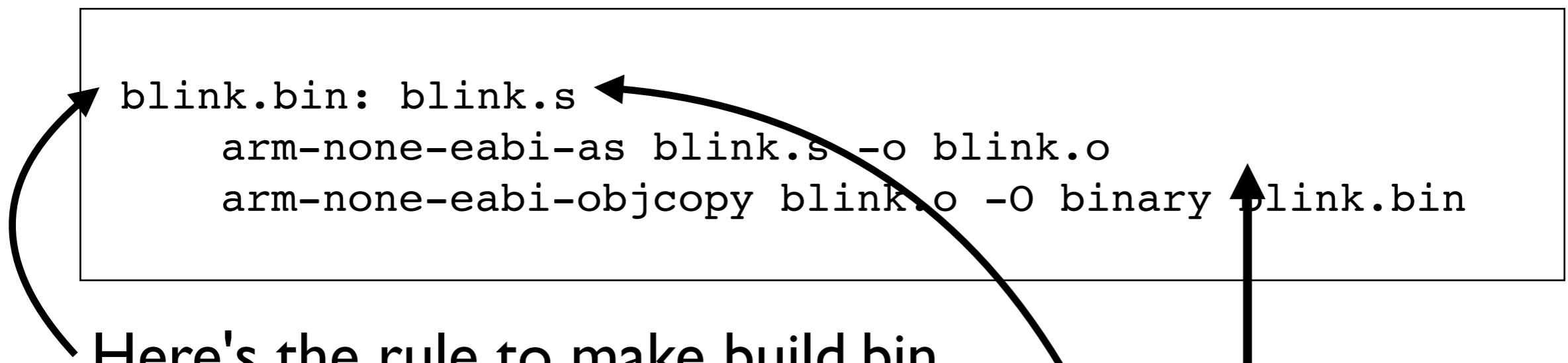
    rm \*.o \*.bin

# Make

Build/compile your code using **make**

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

Rules, targets, dependencies, and recipes



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

# Compiler Explorer

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

The screenshot shows the Compiler Explorer interface. On the left, there is a code editor with the following C code:

```
1 int global = 107;
2
3 void main(void)
4 {
5     global = global + 1;
6 }
```

The code editor has a red box around the language dropdown "C". Above the code editor are two tabs: "C source #1" and "ARM gcc 5.4.1 (none) (Editor #1, Compiler #1) C". The "ARM gcc 5.4.1 (none) (Editor #1, Compiler #1) C" tab is active. To its right is a red box around the compiler dropdown "ARM gcc 5.4.1 (none)" and the optimization flag "-Og". Below the tabs are several output options: "11010", "./.a.out", ".LX0:" (which is checked), "lib.f:", ".text", and "//". A red box highlights ".LX0:". At the bottom, the assembly output is shown:

```
1 main:
2     ldr    r2, .L2
3     ldr    r3, [r2]
4     add    r3, r3, #1
5     str    r3, [r2]
```

<https://godbolt.org>

Configure settings to follow along:

C

ARM gcc 5.4.1 (none)

-Og

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 ?**

```
int i, j;
```

```
i = 1;  
i = 2;  
j = i;
```

```
// can be optimized to
```

```
i = 2;  
j = i;
```

```
// is this ever not equivalent/ok?
```

# **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), such 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.

# 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.