Goals for today

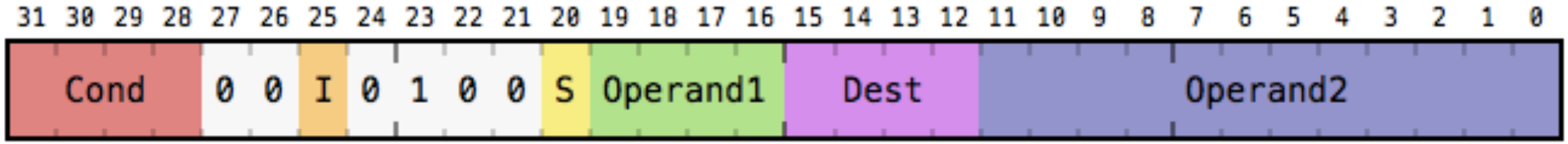
More ARM: condition codes, branches

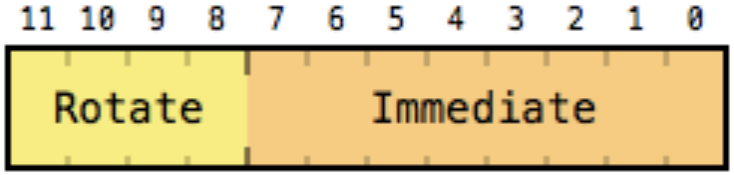
C language as “high-level” assembly

How to compile/build from C source, Makefile

Relationship of C to asm, study translations

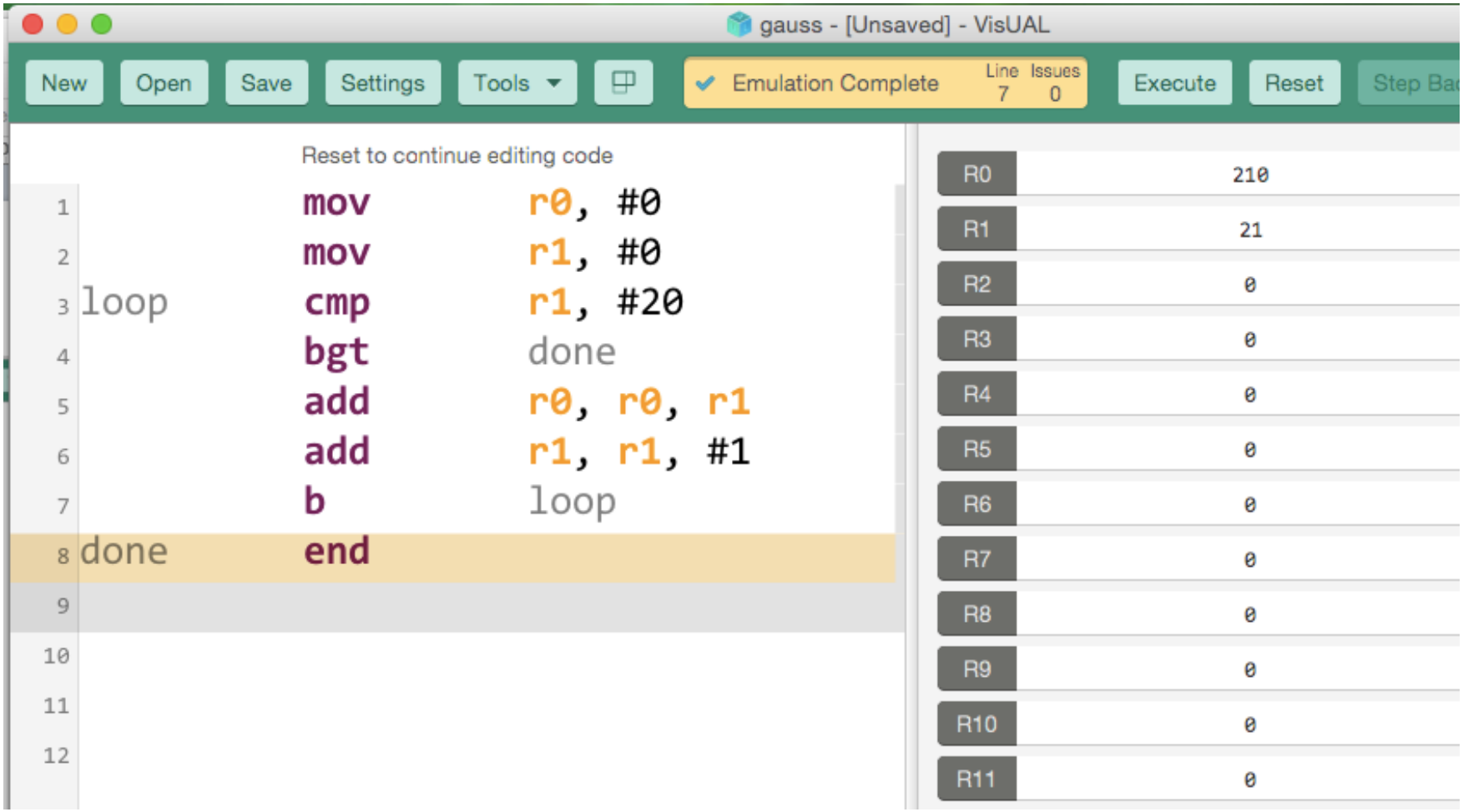
```
# pattern for data processing instructions
#      dst = operand1 op operand2|imm
#
# I - immediate
# S – set condition code
```






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

VisUAL ARM Emulator



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

Condition Codes

Z Result is 0

N Result is < 0

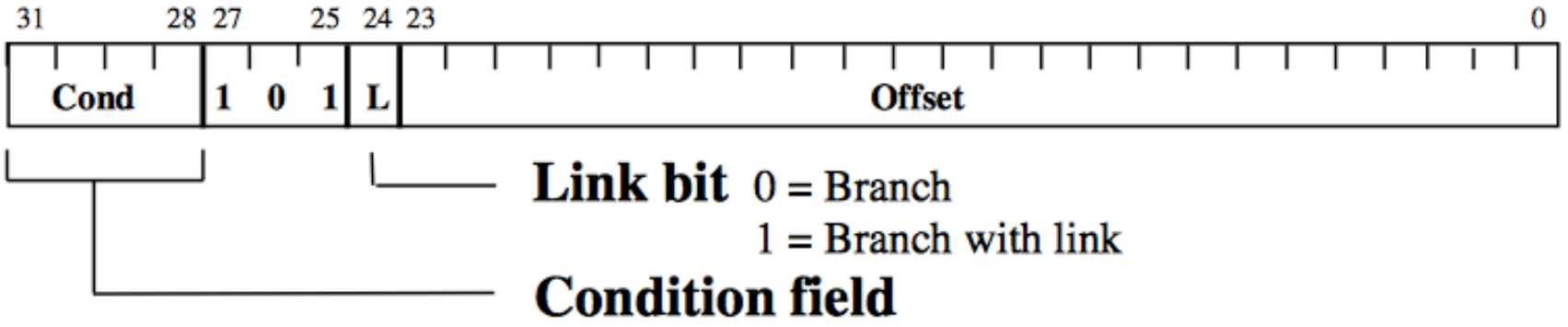
C Carry generated

V Arithmetic overflow

(carry and overflow will be covered later)

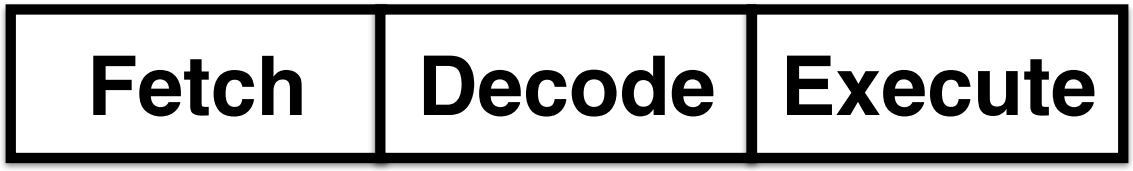
Code	Suffix	Flags	Meaning
0000	EQ	Z set	equal
0001	NE	Z clear	not equal
0010	CS	C set	unsigned higher or same
0011	CC	C clear	unsigned lower
0100	MI	N set	negative
0101	PL	N clear	positive or zero
0110	VS	V set	overflow
0111	VC	V clear	no overflow
1000	HI	C set and Z clear	unsigned higher
1001	LS	C clear 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

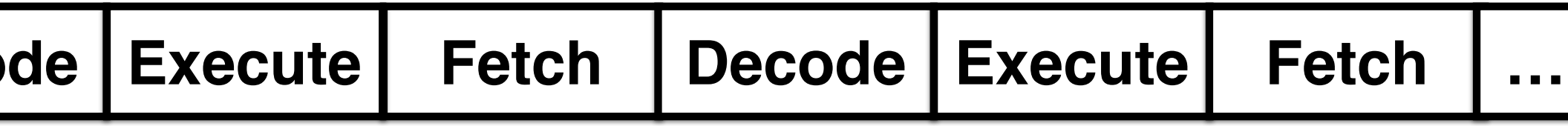



```
b = bal = branch always
1110 101L tttt tttt tttt tttt tttt tttt

bne
0001 101L tttt tttt tttt tttt tttt tttt

branch target expressed PC-relative
blue bits are offset counted in words
```

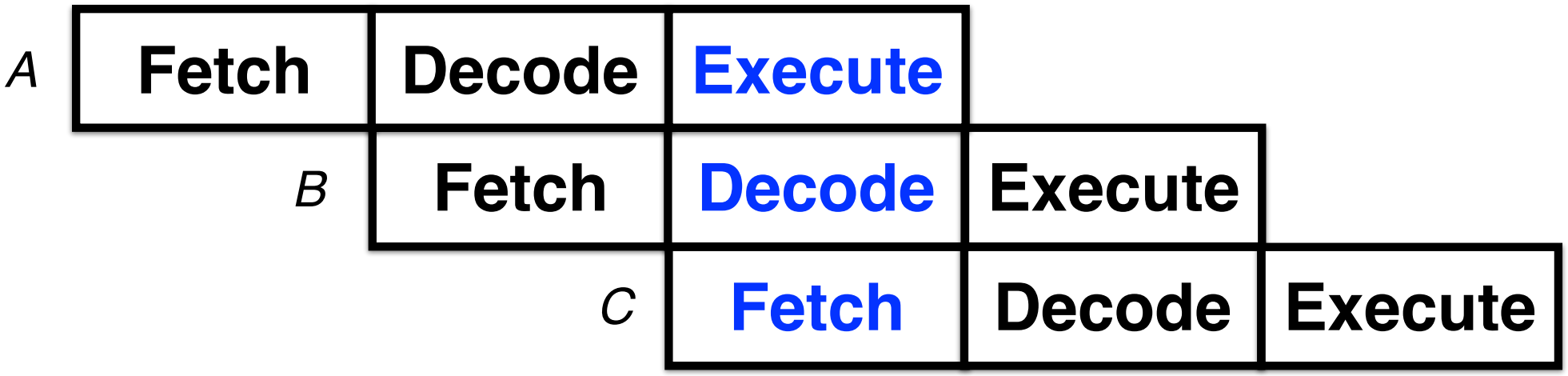
3 steps per instruction



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)

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

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

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

Orthogonal Instructions

Commonalities across operations

Register vs. immediate operands

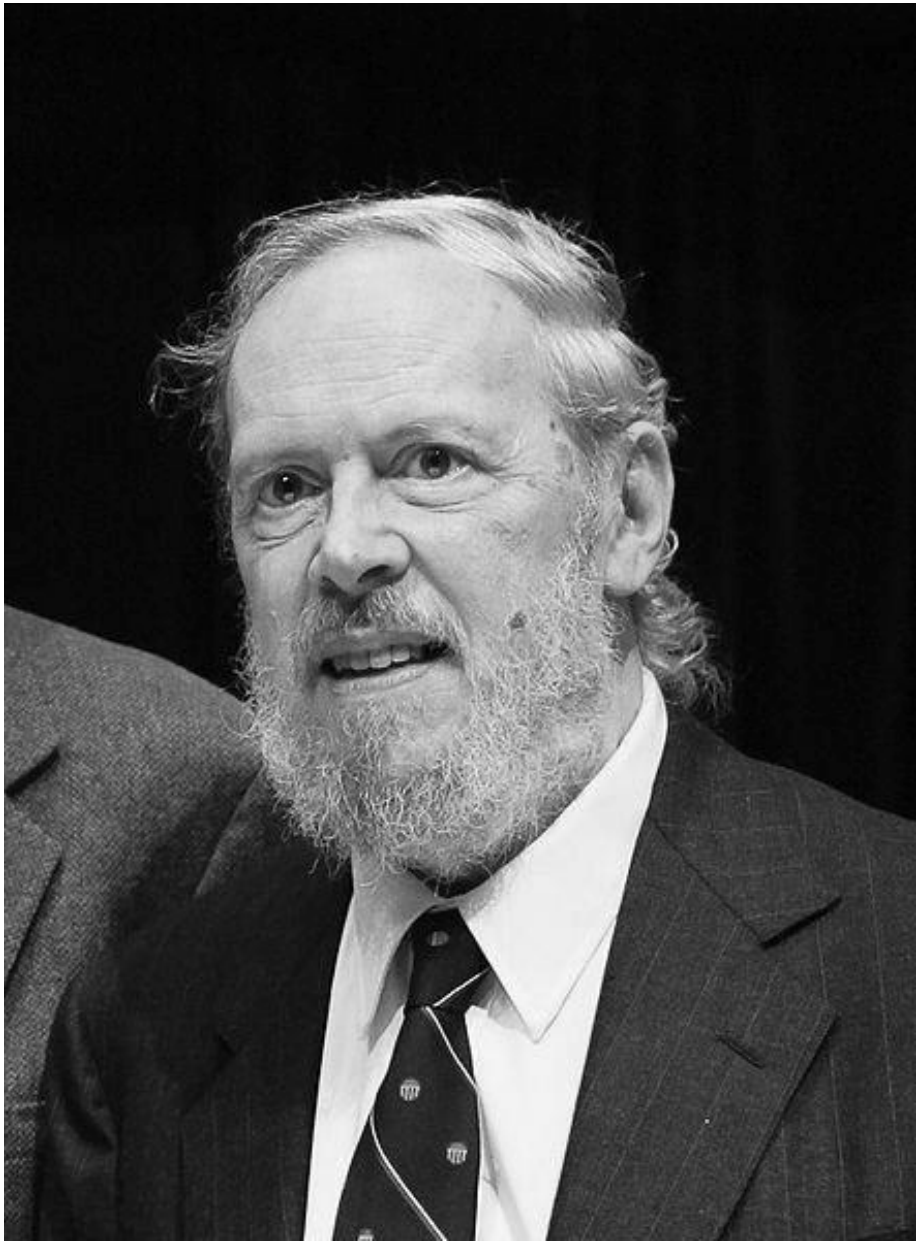
All registers the same**

Predicated/conditional execution

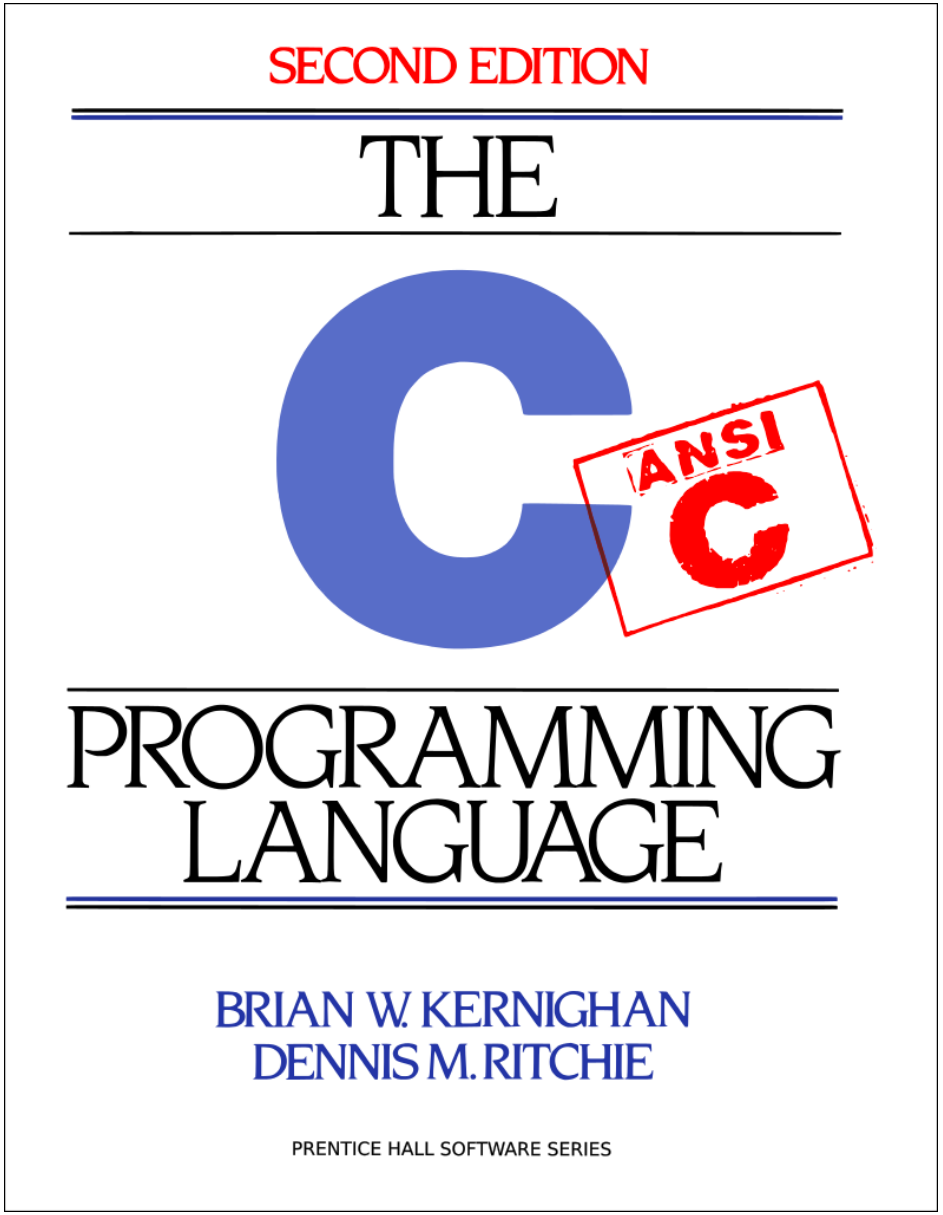
Set condition code (or not)

Orthogonality leads to composability


100 FEET BELOW
SEA LEVEL



Dennis Ritchie



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

```
// 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]
```

Vs.

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

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

Know your tools: make

Makefile is a text file of “recipes”to manage build process

Recipe lists prerequisites and gives sequence of steps to build target from its ingredients

Syntax a bit goopy, whitespace matters

```
ARM = arm-none-eabi
CFLAGS = Og –g -Wall -std=c99 -ffreestanding

con.bin: con.c
        $(ARM)-gcc $(CFLAGS) -c con.c -o con.o
        $(ARM)-objcopy con.o -O binary con.bin
```

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

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 look at translation

https://gcc.godbolt.org

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), these optimizations will not 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 exactly 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 effect 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.