

Admin

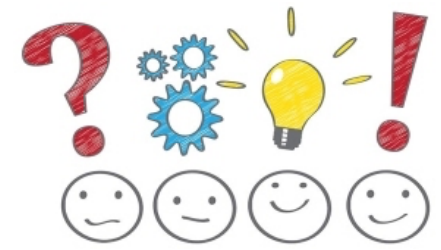
Week 1: RISC-V architecture/assembly

Cycle lecture -> lab -> assign

Please fill out lab I exit form if you missed it

YEAH hours feedback?

Week 2: C control/pointers



Check in

Today: From Assembly to C (and back again)

Branch instructions, PC-relative

C language as “high-level” assembly

What does a compiler do?

Makefiles

Control flow, pc register

Instructions stored in contiguous memory

pc tracks address in memory where to read instruction

pc register separate from x0-x31, not generally accessible, only certain special instructions access/change pc

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

jump instruction assigns pc to different address

j target

Jump is unconditional (always taken)

Branch is conditionally taken based on test

Branch instructions

Mnemonic

Action

BEQ rs1,rs2,imm12

Branch equal ($rs1 = rs2$)

BNE rs1,rs2,imm12

Branch not equal ($rs1 \neq rs2$)

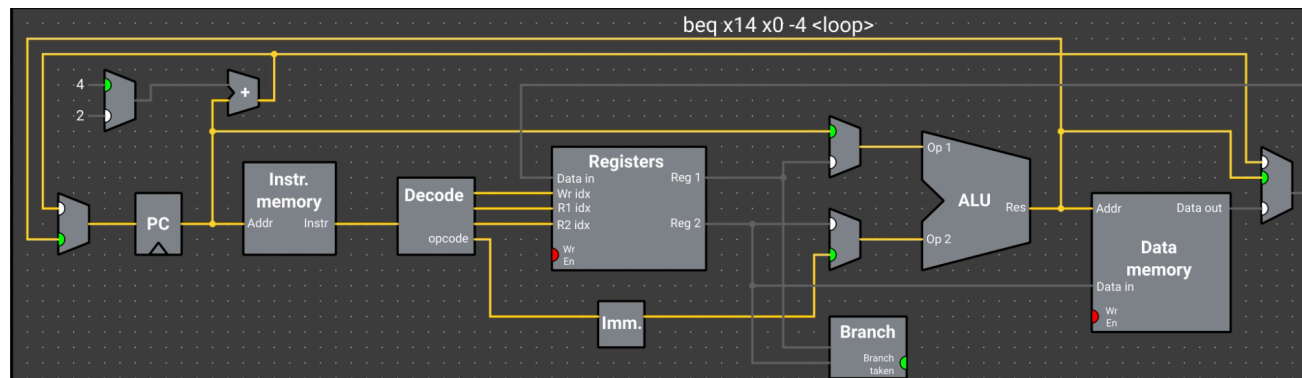
BGE rs1,rs2,imm12

Branch greater than or equal ($rs1 \geq rs2$)

BLT rs1,rs2,imm12

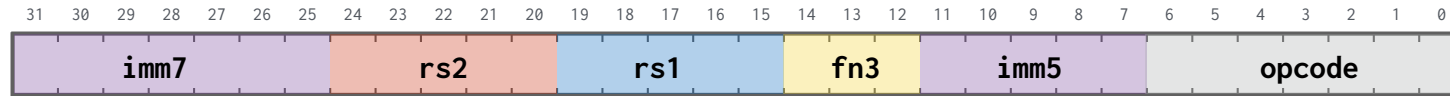
Branch less than ($rs1 < rs2$)

Q: How to... branch greater? Branch less-equal? Branch zero? Branch negative?



If condition satisfied, branch is taken ($pc = pc + imm12$)
otherwise falls through ($pc = pc + 4$)

Branch instruction encoding



`if rs1 cmp_op rs2 pc = pc + imm12`

- branch target computed as PC-relative offset
- purple bits encode offset (immediate)
- "position-independent" code

12-bit immediate expressed as count of 2-byte steps

Q: How far can this reach?

ISA design is an art form!

As much about what is **omitted** as what is **included**

Reduce/simplify: Eliminate redundancies, registers all general-purpose, memory access only through load/store, single addressing mode

Abstraction: Isolate architecture from implementation, no delay slots
branch/load, no condition codes

Regularity: all instructions 4-bytes (2-byte compressed extension), same placement of bits in encoding for ease of decode, common data paths

Modular, extensible: tiny base ISA, optional additions design to be orthogonal, room for growth

Data-informed design: learn from past, decisions backed by "receipts"

Why assembly?

What you see is what you get

No surprises

Precise control, timing

Unfettered access to hardware

But... *tedious, hard to read, hardware-specific, difficult to port*

Why C?

More concise

Easier to read

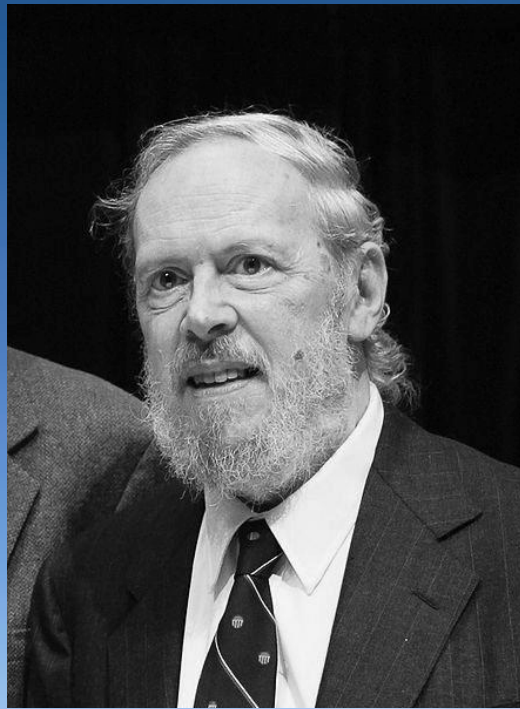
Names for variables and data types

Type-checking

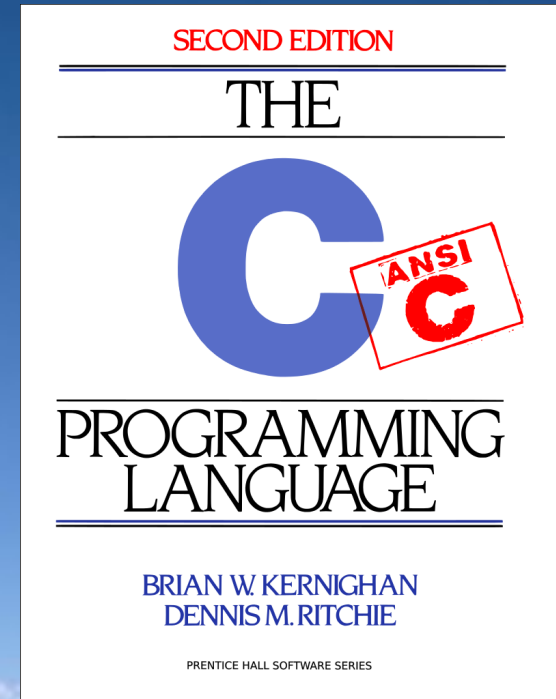
Portable, architecture-neutral

Higher-level abstractions (functions, user-defined types)

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



Dennis Ritchie



C is 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*

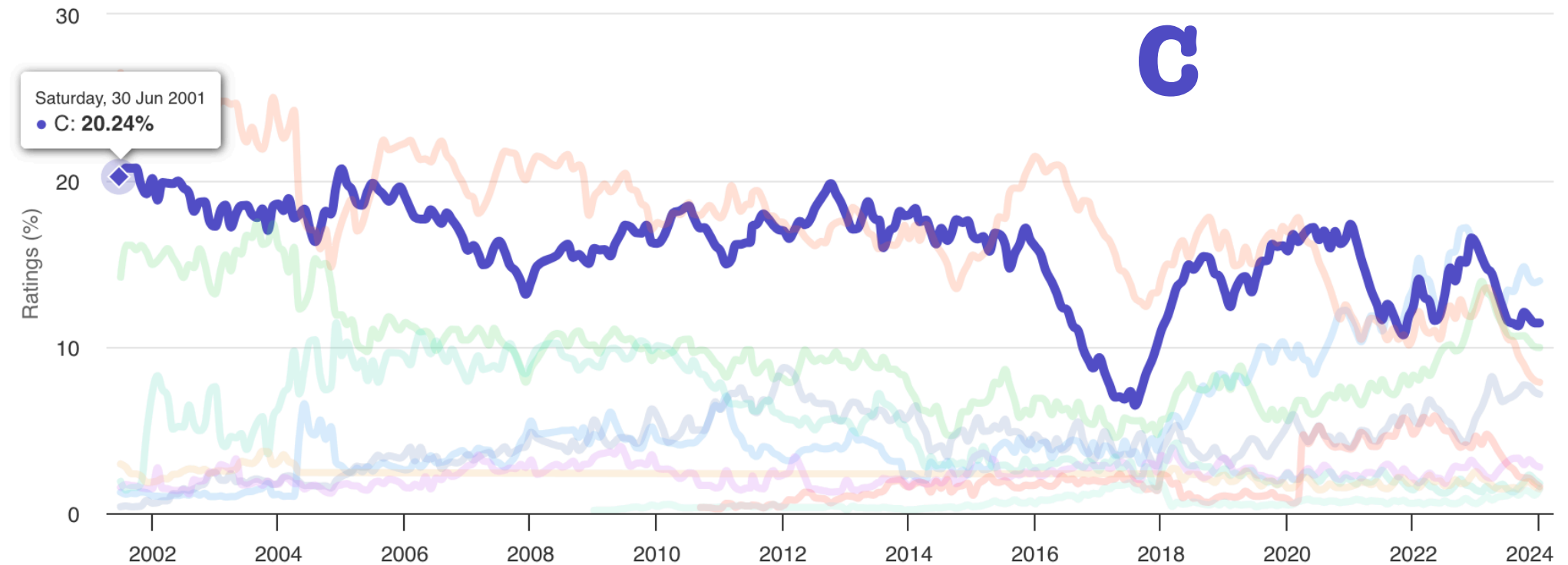
“C: A language that combines all the elegance and power of assembly language with all the readability and maintainability of assembly language”

— *Unknown*

Language popularity over time

TIOBE Programming Community Index

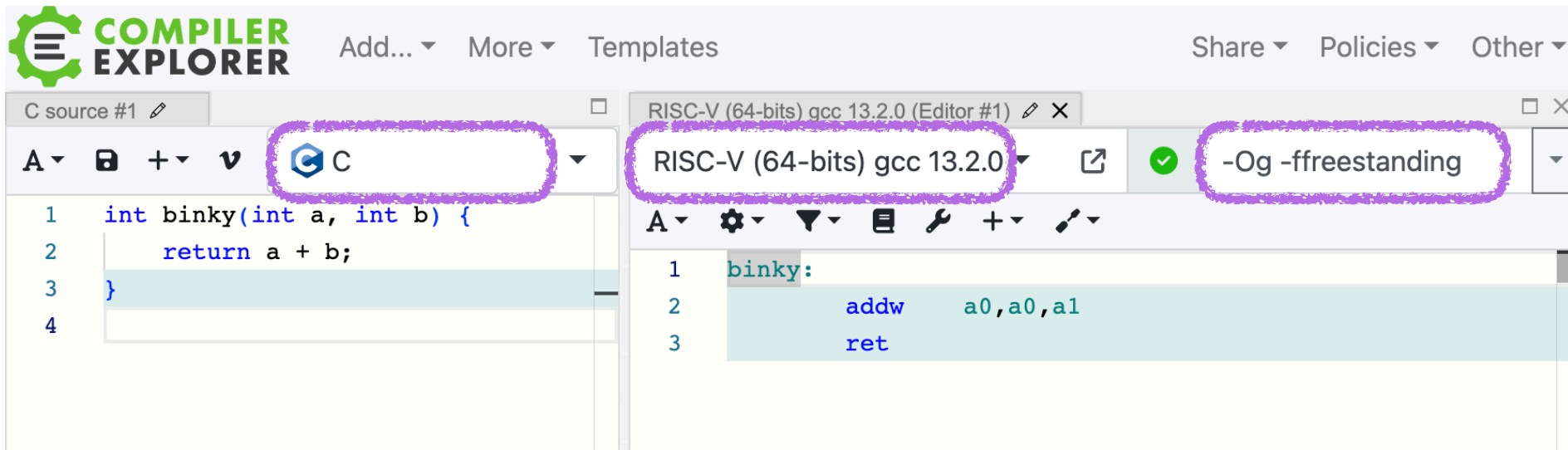
Source: www.tiobe.com



Compiler Explorer

Neat interactive tool to see translation from C to assembly.

Let's try it now!



<https://godbolt.org>

Configure settings to follow along:

C
RISC-V (64 bits) gcc 13.2
-Og -ffreestanding

Major props to the C compiler

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!

Compile-time vs. runtime

Compile-time: compiler running on your laptop

- read C source text, parse/check semantically valid
- analyze code to understand structure/intent
- generate assembly instructions, assembler to binary

Runtime: program binary running on Pi

- load machine instructions to memory
- fetch/decode/execute

Optimizer does work at CT to streamline count of instructions to be executed at RT

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

One-step build process using make

Makefile is text file that describes build steps as "recipes"

Dependencies determine which steps needed to re-build

Rule

`blink.bin: blink.s`

Recipe

```
riscv64-unknown-elf-as blink.s -o blink.o  
riscv64-unknown-elf-objcopy blink.o -O binary blink.bin
```

Target

`run: blink.bin`

Dependency

`mango-run blink.bin`

Writing out explicit recipes becomes onerous, so make has all kinds of ways to match patterns, define variables, etc.

Make pattern rules

NAME = myprogram

ARCH = -march=rv64im -mabi=lp64

CFLAGS = \$(ARCH) -g -Og -Wall -ffreestanding

LDFLAGS = \$(ARCH) -nostdlib

all: \$(NAME).bin

%.bin: %.elf

riscv64-unknown-elf-objcopy \$< -O binary \$@

%.elf: %.o

riscv64-unknown-elf-gcc \$(LDFLAGS) \$< -o \$@

%.o: %.c

riscv64-unknown-elf-gcc \$(CFLAGS) -c \$< -o \$@

Bare-metal vs. Hosted

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

Write our own code for all libs and start files

This puts us in an exclusive club...

```
int getRandomNumber()
{
    return 4; // chosen by fair dice roll.
              // guaranteed to be random.
}
```