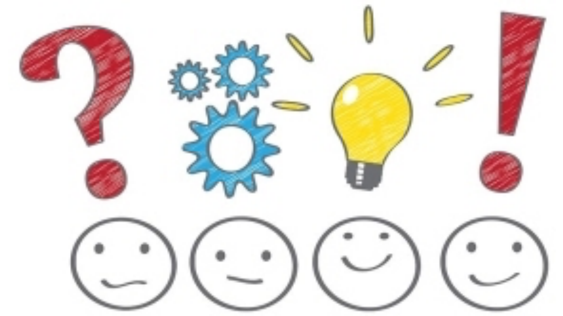
Admin

Weekly cycle: lectures → lab → assign

- Lab release after Monday lecture
  - Pre-lab to prepare
- Assign release after Wed lab



Check in

Today: From Assembly to C
(and back again)

(First cover GPIO content that didn't make into last Friday)

C language as "high-level" assembly

What does a compiler do?

Makefiles

RISC-V in a nutshell

RISC-V Instruction-Set

Erik Engheim <erik.engheim@ma.com>

Arithmetic Operation

Mnemonic	Instruction	Type	Description
`ADD rd, rs1, rs2`	Add	R	`rd ← rs1 + rs2`
`SUB rd, rs1, rs2`	Subtract	R	`rd ← rs1 - rs2`
`ADDI rd, rs1, imm12`	Add immediate	I	`rd ← rs1 + imm12`
`SLT rd, rs1, rs2`	Set less than	R	`rd ← rs1 < rs2 ? 1 : 0`
`SLTI rd, rs1, imm12`	Set less than immediate	I	`rd ← rs1 < imm12 ? 1 : 0`
`SLTU rd, rs1, rs2`	Set less than unsigned	R	`rd ← rs1 < rs2 ? 1 : 0`
`SLTIU rd, rs1, imm12`	Set less than immediate unsigned	I	`rd ← rs1 < imm12 ? 1 : 0`
`LUI rd, imm20`	Load upper immediate	U	`rd ← imm20 << 12`
`AUIP rd, imm20`	Add upper immediate to PC	U	`rd ← PC + imm20 << 12`

Logical Operations

Mnemonic	Instruction	Type	Description
`AND rd, rs1, rs2`	AND	R	`rd ← rs1 & rs2`
`OR rd, rs1, rs2`	OR	R	`rd ← rs1 | rs2`
`XOR rd, rs1, rs2`	XOR	R	`rd ← rs1 ^ rs2`
`ANDI rd, rs1, imm12`	AND immediate	I	`rd ← rs1 & imm12`
`ORI rd, rs1, imm12`	OR immediate	I	`rd ← rs1 | imm12`
`XORI rd, rs1, imm12`	XOR immediate	I	`rd ← rs1 ^ imm12`
`SLL rd, rs1, rs2`	Shift left logical	R	`rd ← rs1 << rs2`
`SRL rd, rs1, rs2`	Shift right logical	R	`rd ← rs1 >> rs2`
`SRA rd, rs1, rs2`	Shift right arithmetic	R	`rd ← rs1 >> rs2`
`SLLI rd, rs1, shamt`	Shift left logical immediate	I	`rd ← rs1 << shamt`
`SRLI rd, rs1, shamt`	Shift right logical imm.	I	`rd ← rs1 >> shamt`
`SRAI rd, rs1, shamt`	Shift right arithmetic immediate	I	`rd ← rs1 >> shamt`

Load / Store Operations

Mnemonic	Instruction	Type	Description
`LD rd, imm12(rs1)`	Load doubleword	I	`rd ← mem[rs1 + imm12]`
`LW rd, imm12(rs1)`	Load word	I	`rd ← mem[rs1 + imm12]`
`LH rd, imm12(rs1)`	Load halfword	I	`rd ← mem[rs1 + imm12]`
`LB rd, imm12(rs1)`	Load byte	I	`rd ← mem[rs1 + imm12]`
`LWU rd, imm12(rs1)`	Load word unsigned	I	`rd ← mem[rs1 + imm12]`
`LHU rd, imm12(rs1)`	Load halfword unsigned	I	`rd ← mem[rs1 + imm12]`
`LBU rd, imm12(rs1)`	Load byte unsigned	I	`rd ← mem[rs1 + imm12]`
`SD rs2, imm12(rs1)`	Store doubleword	S	`rs2 → mem[rs1 + imm12]`
`SW rs2, imm12(rs1)`	Store word	S	`rs2(31:0) → mem[rs1 + imm12]`
`SH rs2, imm12(rs1)`	Store halfword	S	`rs2(15:0) → mem[rs1 + imm12]`
`SB rs2, imm12(rs1)`	Store byte	S	`rs2(7:0) → mem[rs1 + imm12]`

Branching

Mnemonic	Instruction	Type	Description
`BEQ rs1, rs2, imm12`	Branch equal	SB	`if rs1 = rs2` `pc ← pc + imm12`
`BNE rs1, rs2, imm12`	Branch not equal	SB	`if rs1 ≠ rs2` `pc ← pc + imm12`
`BGE rs1, rs2, imm12`	Branch greater than or equal	SB	`if rs1 ≥ rs2` `pc ← pc + imm12`
`BGEU rs1, rs2, imm12`	Branch greater than or equal unsigned	SB	`if rs1 >= rs2` `pc ← pc + imm12`
`BLT rs1, rs2, imm12`	Branch less than	SB	`if rs1 < rs2` `pc ← pc + imm12`
`BLTU rs1, rs2, imm12`	Branch less than unsigned	SB	`if rs1 < rs2` `pc ← pc + imm12 << 1`
`JAL rd, imm20`	Jump and link	UJ	`rd ← pc + 4` `pc ← pc + imm20`
`JALR rd, imm12(rs1)`	Jump and link register	I	`rd ← pc + 4` `pc ← rs1 + imm12`

Pseudo Instructions

Mnemonic	Instruction	Base instruction(s)
`LI rd, imm12`	Load immediate (near)	`ADDI rd, zero, imm12`
`LI rd, imm`	Load immediate (far)	`LUI rd, imm[31:12]` `ADDI rd, rd, imm[11:0]`
`LA rd, sym`	Load address (far)	`AUIPC rd, sym[31:12]` `ADDI rd, rd, sym[11:0]`
`MV rd, rs`	Copy register	`ADDI rd, rs, 0`
`NOT rd, rs`	One's complement	`XORI rd, rs, -1`
`NEG rd, rs`	Two's complement	`SUB rd, zero, rs`
`BGT rs1, rs2, offset`	Branch if rs1 > rs2	`BLT rs2, rs1, offset`
`BLE rs1, rs2, offset`	Branch if rs1 ≤ rs2	`BGE rs2, rs1, offset`
`BGTU rs1, rs2, offset`	Branch if rs1 > rs2 (unsigned)	`BLTU rs2, rs1, offset`
`BLEU rs1, rs2, offset`	Branch if rs1 ≤ rs2 (unsigned)	`BGEU rs2, rs1, offset`
`BEQZ rs1, offset`	Branch if rs1 = 0	`BEQ rs1, zero, offset`
`BNEZ rs1, offset`	Branch if rs1 ≠ 0	`BNE rs1, zero, offset`
`BGEZ rs1, offset`	Branch if rs1 ≥ 0	`BGE rs1, zero, offset`
`BLEZ rs1, offset`	Branch if rs1 ≤ 0	`BGE zero, rs1, offset`
`BGTZ rs1, offset`	Branch if rs1 > 0	`BLT zero, rs1, offset`
`J offset`	Unconditional jump	`JAL zero, offset`
`CALL offset12`	Call subroutine (near)	`JALR ra, ra, offset12`
`CALL offset`	Call subroutine (far)	`AUIPC ra, offset[31:12]` `JALR ra, ra, offset[11:0]`
`RET`	Return from subroutine	`JALR zero, 0(ra)`
`NOP`	No operation	`ADDI zero, zero, 0`

32-bit instruction format

	31 30 29 28 27 26 25	24 23 22 21 20	19 18 17 16 15	14 13 12	11 10 9 8 7	6 5 4 3 2 1 0
R	func	rs2	rs1	func	rd	opcode
I	immediate		rs1	func	rd	opcode
SB	immediate	rs2	rs1	func	immediate	opcode
UJ	immediate				rd	opcode

Register File

r0	r1	r2	r3
r4	r5	r6	r7
r8	r9	r10	r11
r12	r13	r14	r15
r16	r17	r18	r19
r20	r21	r22	r23
r24	r25	r26	r27
r28	r29	r30	r31

Register Aliases

zero	ra	sp	gp
tp	t0	t1	t2
s0/fp	s1	a0	a1
a2	a3	a4	a5
a6	a7	s2	s3
s4	s5	s6	s7
s8	s9	s10	s11
t3	t4	t5	t6

ra - return address
sp - stack pointer
gp - global pointer
tp - thread pointer

t0 - t6 - Temporary registers
s0 - s11 - Saved by callee
a0 - 17 - Function arguments
a0 - a1 - Return value(s)

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?

<u>What you see is what you get</u>

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

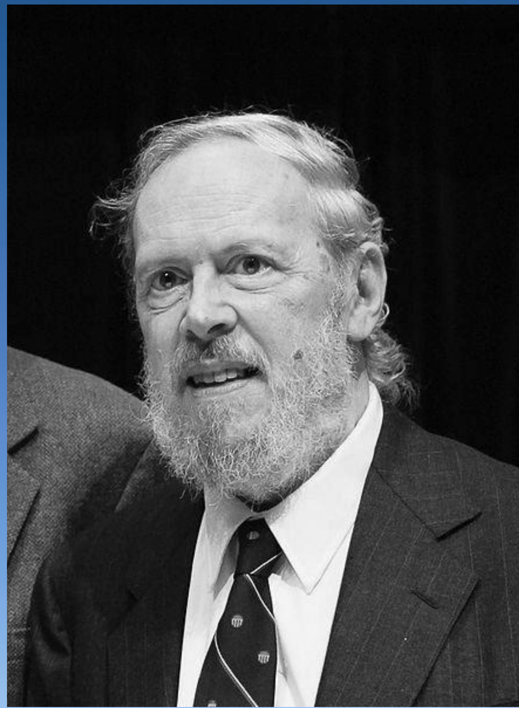
Names for variables and data types

Type-checking

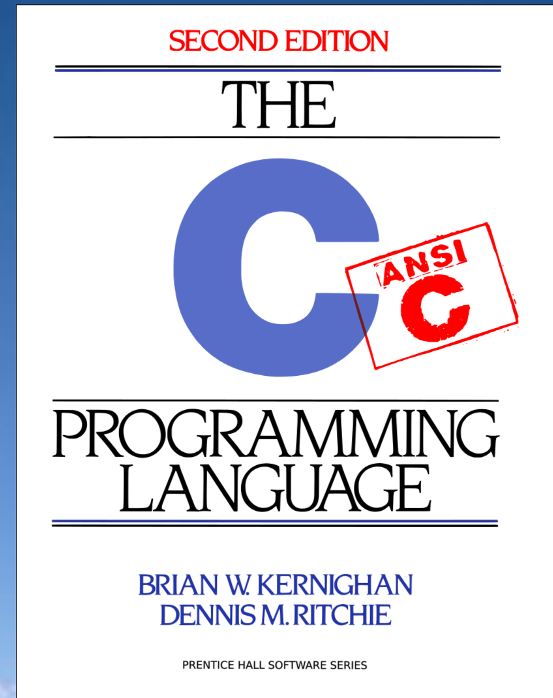
Portable, architecture-neutral

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

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*

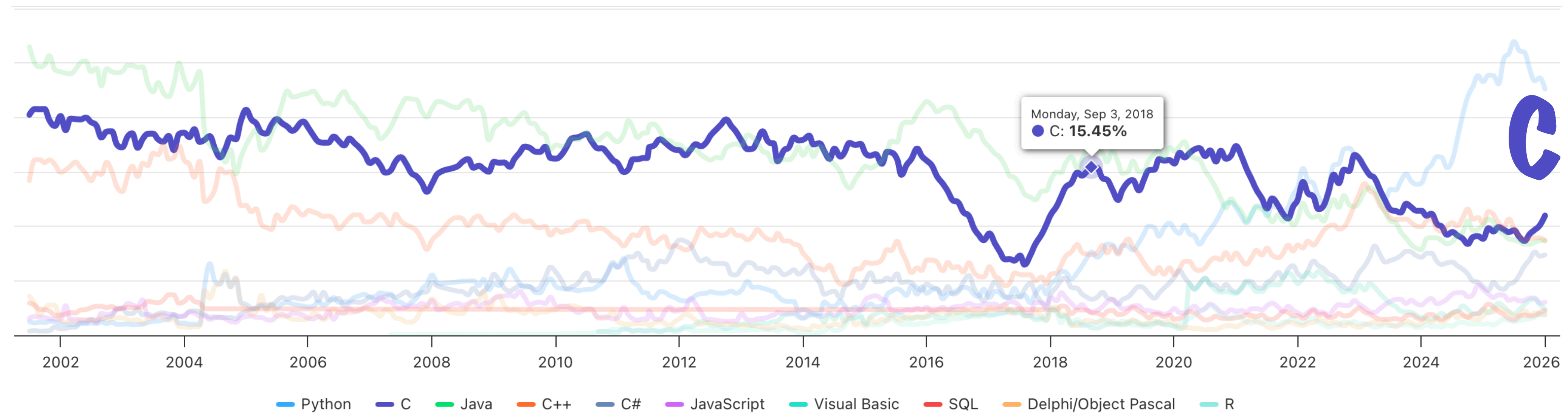
The C Programming Language

"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



Know your tools: **assembler**

The *assembler* reads assembly instructions (text) and outputs as machine-code (binary). This translation is mechanical and fully deterministic.

```
$ riscv64-unknown-elf-as blink.s -o blink.o
$ riscv64-unknown-elf-objcopy blink.o -O binary blink.bin
$ hexdump -C blink.bin
37 05 00 02 93 05 10 00  ...
```

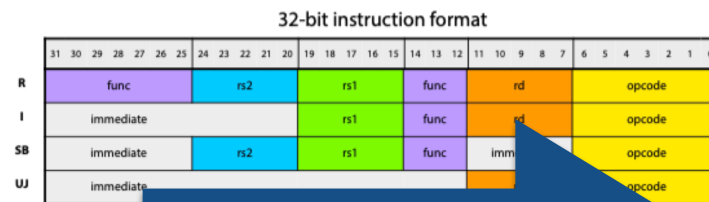
```
    lui     a0,0x2000
    addi    a1,zero,1
    sw      a1,0x30(a0)

loop:
    xori    a1,a1,1
    sw      a1,0x40(a0)

    lui     a2,0x3f00
delay:
    addi    a2,a2,-1
    bne     a2,zero,delay

    j       loop
```

blink.s




```
37 05 00 02 93 05 10 00
23 28 b5 02 93 c5 15 00
23 20 b5 04 37 06 f0 03
13 06 f6 ff e3 1e 06 fe
6f f0 df fe
```

blink.bin

Know your tools: **compiler**

The *compiler* reads C source (text) and translates to assembly instructions (assembler used to convert to binary from there)

```
int sum(int a, int b) {
    return a + b;
}
```

```
sum:
    addw   a0,a0,a1
    ret
```

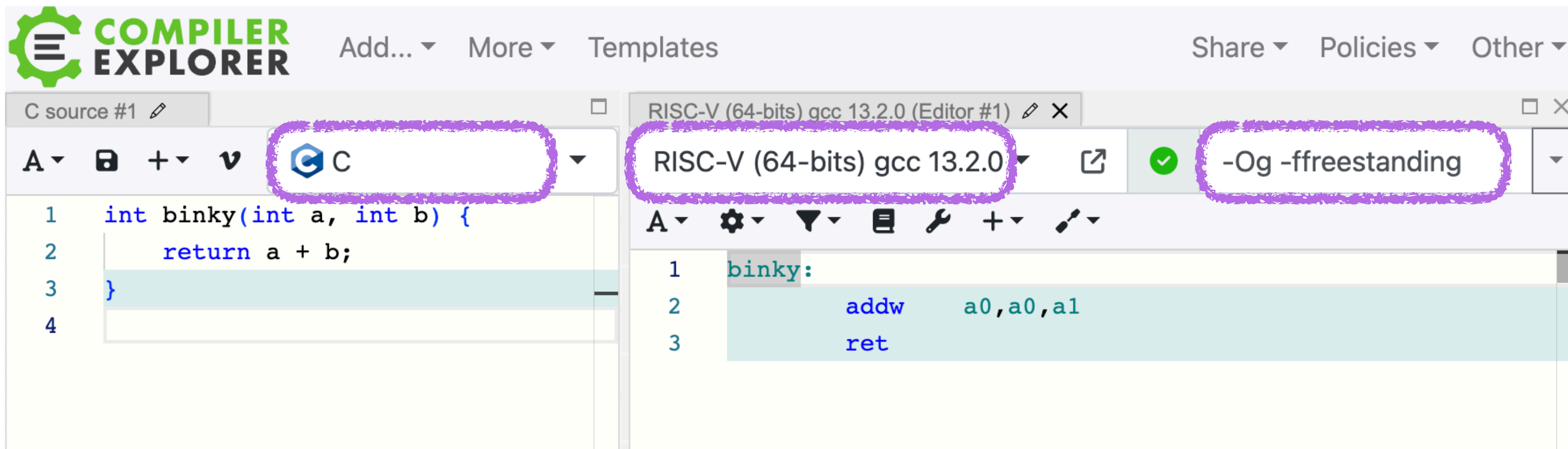
Tokenize → Parse → Semantic analysis → Code generation

This translation is complex, high artistry

Compiler Explorer

Neat interactive translation from C to assembly

Follow along as we 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

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

Recipe

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

run: blink.bin
    mango-run blink.bin
```

Target

Dependency

Writing explicit recipes for every file is onerous,
Use **`make`** match by pattern to create general rules

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-ld $(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.
}
```