# ДОКТОР, У МЕНЯ НОВАЯ ПЛАТФОРМА!

АНТОН КОРОБЕЙНИКОВ

# Platform Support?

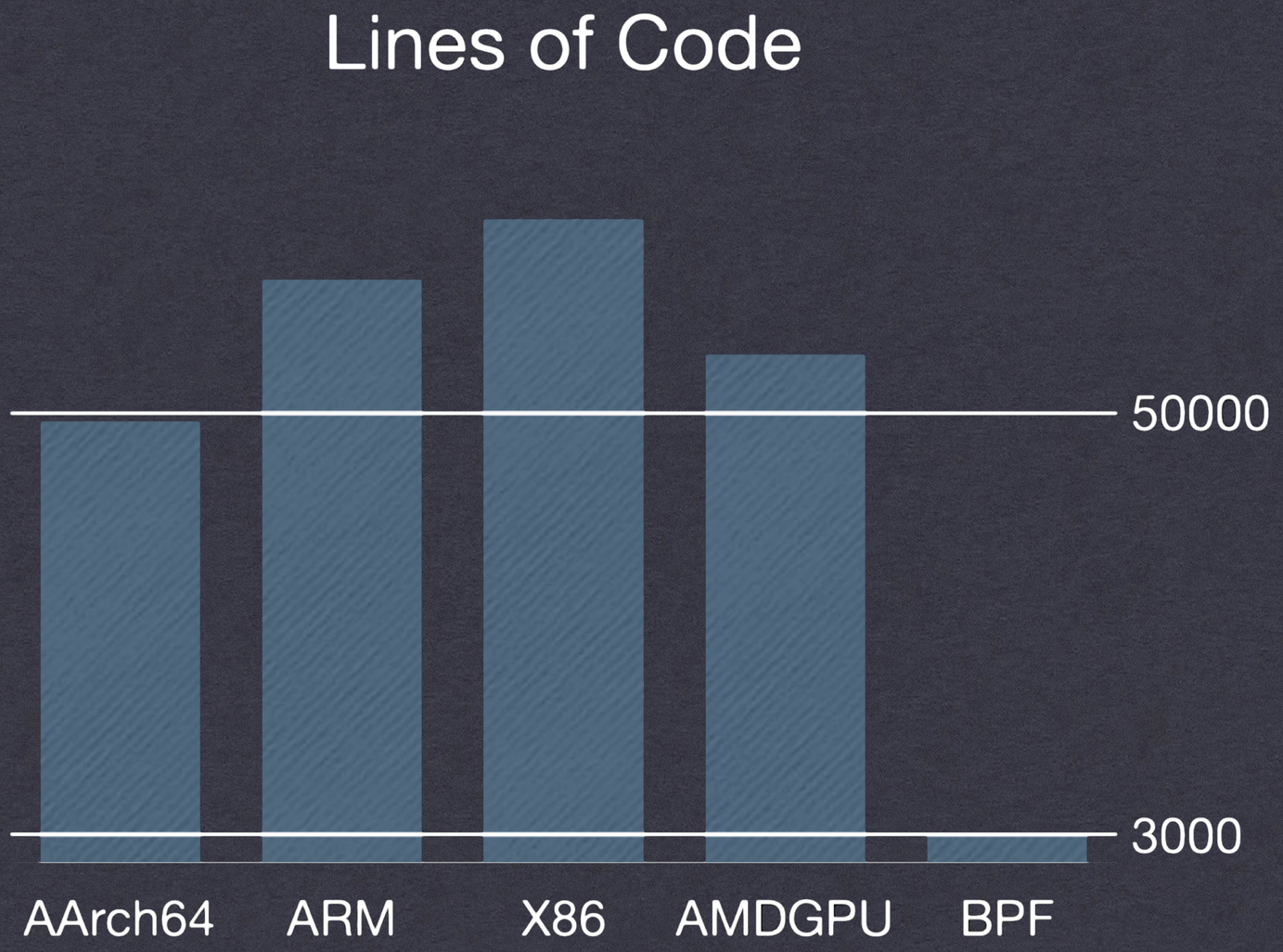
- Calling convention(s)
- TLS, DSO, Platform ABI, …
- Target triples & subtargets
- Frontend(s)
- Backend

# Backend: generic things

- Always in-tree
- Mainline vs downstream?
- Private changes
- Plan to sync with LLVM mainline

# How large a typical backend is?

# How large a typical backend is?



# How to make a backend?

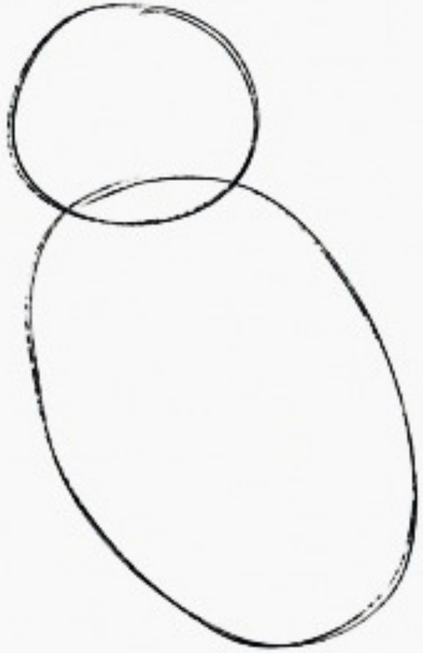


Fig 1. Draw two circles

Fig 2. Draw the rest of the damn Owl

# Two approaches:

- Assembler / Disassembler
- “Hello world”

# Compilation Flow

Compiler:

C/C++ → LLVM IR → Instruction Selector → MIR → MC → .o

Assembler:

.s → MC → .o

# Assembler / Disassembler

- Common part in compilation flows: MC layer
- Encode / decode machine instructions
- Produce perfectly fine ELF

# MC layer: key ingredients

- Describe instruction encoding and assembly syntax
- Describe registers and other instruction operands
- Assembly parsing (can be skipped)
- Necessary backend boilerplate
- Tests

# TableGen: the Great and Powerful

- DSL used to describe different aspects of target (not only)
- Language itself is simple, but this is not enough
- Lots of tablegen “backends” that generate code out of descriptions

# Boilerplate

- Directory inside lib/Target: lib/Target/Foo
- Build system: CMakeLists.txt
- Target registration
- Triple parsing
- Test infrastructure: lit + FileCheck

# “Hello world” approach

- Start from the small code snippets
- Iteratively work over testcases covering more and more
- Add instruction definitions as necessary

# “Hello world” approach

- Start from the small code snippets
- Iteratively work over testcases covering more and more
- Add instruction definitions as necessary

```
define void @f() {
    ret void
}
```

# “Hello world” approach

- Start from the small code snippets
- Iteratively work over testcases covering more and more
- Add instruction definitions as necessary

```
define void @f() {
    ret void
}
```

```
define i32 @double() {
    ret i32 42
}
```

# “Hello world” approach

- Start from the small code snippets
- Iteratively work over testcases covering more and more
- Add instruction definitions as necessary

```
define i32 @double(i32 %x) {
    %y = add i32 %x, %x
    ret i32 %y
}
```

# Important choice in year 2023

# Important choice in year 2023

SelectionDAG or GlobalISel?

# Important choice in year 2023

- SelectionDAG:
  - Mature
  - Lots of code and examples
  - Has its own limitations (per BB)
- GlobalISel
  - Fresh and shiny (well, not quite)
  - Can work cross-BB
  - Might have some quirks especially for optimized code

# More Decisions

- Instruction scheduling?
- Target-specific optimisations, peepholes
- HW loops
- …

# Custom Changes

- Try to generalize
- Do it as you'd submit it to LLVM mainline tomorrow
- Try to make isolated changes: hooks, passes, etc.
- Do not: add hacks & kludges here and there

# Where to get help?

- Other backends:
  - Small: BPF, RISC-V, MSP430
  - Larger: AArch64
- Generic code, parent classes
- Some docs on llvm.org/docs (patches are welcome!)
- Discourse

# Q & A