从零理解一个 x86-64 → ARM64 二进制翻译器

写给汇编初学者的逐步详解

rousaoe 项目文档

2026-06-22

摘要

本文档面向**完全没有汇编基础**的读者，目标是把一件听起来很”黑魔法”的事情讲清楚：让一个为 x86 处理器编译的 Linux 程序，在 ARM 处理器的 Linux 上跑起来。我们会从”CPU 到底在做什么””寄存器是什么”这种最基础的概念讲起，逐条解释一个真实小程序里的每一条汇编指令，再逐字节拆开它编译出来的机器码；然后一步步讲我们这个最小可用翻译器（MVP）的每个部件——加载器、解码器、解释器、系统调用层，以及真正把 x86 指令**翻译成 ARM 机器码**的 JIT。每一步都配有真实代码与真实运行结果。

目录

1 本文档怎么读

这份文档是”边讲概念边看代码”。你会看到三类排版：

- **正文**：用大白话解释概念，尽量不假设你懂任何术语。
- **代码块**：带行号、灰底，是项目里**真实存在**的代码或真实的命令输出。
- **提示框**：蓝色【**知识点**】补充背景；红色【**注意**】标出容易踩的坑。

阅读建议

如果你是初学者，建议按顺序读第 2 到第 7 节（概念与汇编基础），再读第 8 节往后的代码部分。如果你已经懂汇编，可以直接从第 7 节（总体架构）开始。

读完你应该能回答这些问题：CPU 怎么执行指令？一条 `mov` 到底干了什么？机器码里那串 `0x48 c7 c0` 是怎么来的？为什么 ARM 不能直接跑 x86？我们是怎么”翻译”的？什么是 JIT？

2 引言：为什么 x86 程序不能直接在 ARM 上运行

2.1 程序最终是一串”指令”

你写的 C 代码、Python 代码，最终都要变成 CPU 能懂的东西才能运行。CPU 能直接懂的，是一串 二**进制的机器指令**。每一种 CPU 都有自己规定好的”指令表”——规定了”哪几个字节代表加法”“哪几个字节代表跳转”。这张指令表就叫**指令集架构**（ISA, Instruction Set Architecture）。

ISA 是什么

ISA 是 CPU 与软件之间的”合同”：它规定了有哪些寄存器、有哪些指令、每条指令用哪些字节表示、执行后会产生什么效果。常见的 ISA 有两大阵营：**x86 / x86-64**（Intel、AMD 的台式机/服务器芯片）和 **ARM / ARM64（AArch64）**（手机、苹果 M 系列、很多服务器）。

2.2 两种 CPU 的”语言”不一样

问题就出在这里：x86 程序里那串字节，是按 **x86 的指令表**编码的；而 ARM 的 CPU 拿到这串字节，会按 **ARM 的指令表**去解释——结果完全是乱码。这就像把一篇用中文写的文章，逐字”念”给只懂法语的人听：同样的符号，规则不同，意义就全错了。

举个具体例子：x86-64 里”系统调用”这条指令编码成两个字节 `0x0f 05`。同样这两个字节在 ARM64 上根本不是这个意思。所以**不翻译，就没法跑**。

2.3 三条翻译路线，我们选哪条

让一个 ISA 的程序在另一个 ISA 上跑，业界有三条路：

方式	做法	特点
解释执行	像翻译官一样，读一条 x86 指令、当场” 模拟” 它的效果，再读下一条	简单、正确，但慢（一条顶几十条）
静态翻译	运行前把整个 x86 程序一次性翻成 ARM 程序	快，但遇到” 运行时才知道往哪跳” 就抓瞎
动态翻译（DBT）	程序**边跑边翻**：执行到哪段就把那段翻成 ARM 机器码，翻完缓存起来	本项目采用；既能处理复杂跳转，又能优化

DBT（Dynamic Binary Translation，动态二进制翻译）是 QEMU、苹果 Rosetta 2、Box64、FEX-Emu 等成熟项目共同的选择。本项目 `rousaoe` 就是从零实现一个最小的 DBT。

我们站在哪一层

我们做的是**用户态**翻译：不模拟整台电脑（不模拟硬盘、网卡、内存管理单元），只翻译程序自己的指令，并在程序想” 调用操作系统” 时，把这个请求转交给真正的 ARM Linux 内核。这比模拟整机简单得多，也是 QEMU-user / Box64 / FEX 的做法。

3 预备知识：CPU、内存、寄存器、机器码、汇编

这一节给零基础读者。已经懂的可以跳到第 4 节。

3.1 CPU 在做的事，其实很” 笨”

CPU 的核心工作循环简单到出奇，就三步，无限重复：

1. **取指**（fetch）：从内存里读出下一条指令的字节。
2. **译码**（decode）：看懂这串字节是什么指令、操作哪些数据。
3. **执行**（execute）：真正去做（加法、搬数据、跳转……），然后指向下一条。

” 下一条指令在哪” 由一个特殊寄存器记录，在 x86-64 上叫 `rip`（Instruction Pointer，指令指针）。每执行完一条，`rip` 就往后挪到下一条。我们的解释器（第 10 节）就是用软件**原样照搬**了这个” 取指-译码-执行” 循环。

3.2 内存：一长条带编号的格子

内存可以想象成一**长排格子**，每个格子放 1 个字节（8 个二进制位），每个格子有一个编号，这个编号就是**地址**。程序的指令、数据，全都按地址摆在内存里。CPU 靠地址去读写内存。地址通常用十六进制写，比如 `0x401000`。

字节、十六进制、大小端

1 字节 = 8 位，能表示 0~255，写成两位十六进制（`0x00~0xff`）。x86-64 是**小端序**（little-endian）：一个多字节的数，**低位字节放在低地址**。例如数值 `0x00000001` 在内存里按字节是 `01 00 00 00`。后面拆机器码时你会反复见到它。

3.3 寄存器：CPU 手边的几个超快小抽屉

去内存取数据相对慢。所以 CPU 内部自带一小撮**超高速的存储位置**，叫**寄存器**。它们数量很少（x86-64 有 16 个通用寄存器），但快得多。CPU 的运算几乎都在寄存器之间进行：把数据从内存搬进寄存器、算、再搬回内存。

3.4 机器码 vs 汇编：同一件事的两种写法

- **机器码**：CPU 真正吃的二进制字节，比如 `48 c7 c0 01 00 00 00`。人看着头大。
- **汇编语言**：机器码的**人类可读别名**，一条汇编对应一条机器指令，比如 `mov rax, 1`（把数字 1 放进寄存器 `rax`）。

汇编器（assembler）负责把汇编翻成机器码；反汇编器（disassembler）反过来把机器码还原成汇编。本项目的**解码器**干的就是反汇编器的活：把 x86 机器字节还原成” 这是哪条指令”。

3.5 用户态、内核态与系统调用

程序自己不能直接操作硬件（屏幕、磁盘）。要” 打印一行字”” 退出程序” 这种事，必须**请操作系统内核帮忙**。程序平时跑在受限的**用户态**；当它需要内核服务时，执行一条特殊指令（x86-64 上是 `syscall`）切到**内核态**，这就叫**系统调用**（syscall）。

系统调用怎么传参

在 x86-64 Linux 上，约定是：寄存器 `rax` 放**系统调用号**（要哪个服务），参数依次放进 `rdi`、`rsi`、`rdx`、`r10`、`r8`、`r9`，然后执行 `syscall`。内核做完，把返回值放回 `rax`。我们的 hello 程序就用了两个系统调用：`write`（号 1，打印）和 `exit`（号 60，退出）。

4 x86-64 的寄存器与系统调用约定

4.1 16 个通用寄存器

x86-64 有 16 个 64 位通用寄存器（GPR, General-Purpose Register）。名字有历史包袱，但你只要记住它们就是 16 个能放整数/地址的” 抽屉”：

编号	寄存器	常见用途（按 Linux 调用约定）
0	`rax`	运算、系统调用号、返回值
1	`rcx`	计数、第 4 个函数参数
2	`rdx`	第 3 个参数 / 系统调用第 3 参数
3	`rbx`	通用（被调用者保存）
4	`rsp`	**栈指针**（指向当前栈顶）
5	`rbp`	栈基址
6	`rsi`	第 2 个参数 / 系统调用第 2 参数
7	`rdi`	第 1 个参数 / 系统调用第 1 参数
8~15	`r8~r15`	扩展通用寄存器

编号顺序不是字母序

注意上表的**编号**：是 rax, rcx, rdx, rbx, rsp, rbp, rsi, rdi……这是 x86 指令编码里规定的顺序，**不是字母顺序**（rcx 是 1、rbx 是 3）。这个顺序非常重要：机器码里用 3 个二进制位（0~7）表示用哪个寄存器，正是按这个编号。我们解码器里那个 enum 就照搬了它。

我们在代码里把这 16 个寄存器存成一个数组 cpu->r[16]，数组下标就是上表编号，所以 cpu->r[7] 就是 rdi。

5 测试程序 hello.S：逐行精讲

我们用来验证翻译器的”小白鼠”是一个**不依赖任何库**（freestanding）的 x86-64 程序，只做两件事：打印一行字、退出。完整源码如下（AT&T 语法）：

Listing 1: tests/hello.S（完整）

```
    .text
    .global _start
_start:
    movq    $1, %rax            /* __NR_write  系统调用号 1 */
    movq    $1, %rdi            /* fd = 1, 标准输出 */
    leaq    msg(%rip), %rsi     /* buf = 字符串地址 */
    movq    $msglen, %rdx       /* count = 字符串长度 */
    syscall                     /* 调用内核: write(1, msg, len) */

    movq    $60, %rax           /* __NR_exit   系统调用号 60 */
    xorq    %rdi, %rdi          /* status = 0 */
    syscall                     /* 调用内核: exit(0) */

    .section .rodata
msg:
    .ascii  "hello from x86-64, translated to arm64!\n"
    .set    msglen, . - msg
```

AT&T 语法的两个约定

这里用的是 AT&T 汇编语法（GNU 汇编器默认）：**(1)** 寄存器前面加 %（如 %rax）；**(2)** 立即数（写死的常数）前面加 $（如 $1 就是数字 1）。还有一种 Intel 语法不加这些前缀。两种只是写法不同，编出来的机器码一样。

下面逐条拆解。

5.1 movq $1, %rax ——把常数装进寄存器

mov 是最常用的指令：**搬运数据**。movq $1, %rax 的意思是”把立即数 1 放进寄存器 rax”。这里把 1 放进 rax，是因为我们准备做系统调用，而 rax 要放**系统调用号**——write 的号正好是

1。

mov 的方向

AT&T 语法是**源在左、目标在右**：`movq $1, %rax` 是把 1 写进 `rax`。（Intel 语法反过来，目标在左。）后缀 `q` 表示操作 64 位（quad word，8 字节）。

5.2 `movq $1, %rdi` ——准备第 1 个参数

同理，把 1 放进 `rdi`。`rdi` 是系统调用的**第 1 个参数**。对 `write` 来说，第 1 个参数是**文件描述符**：1 代表” 标准输出”，也就是终端屏幕。

5.3 `leaq msg(%rip), %rsi` ——算出字符串的地址

这条是初学者最容易困惑的，但它非常重要。`lea` = Load Effective Address，” 加载**有效地址**”。注意：它**不读内存里的内容**，只**计算出一个地址**放进寄存器。

`msg(%rip)` 是一种寻址方式叫 **RIP 相对寻址**：地址 =（下一条指令的地址 `rip`）+（一个固定偏移）。汇编器会算好这个偏移，让最终结果正好等于字符串 `msg` 的地址。执行后 `rsi` 里就是字符串首字节在内存中的地址——这正是 `write` 需要的第 2 个参数 `buf`（要打印的内容在哪）。

lea 与 mov 的关键区别

`lea` 拿的是” 地址本身”，`mov` 从内存拿的是” 地址里存的内容”。打个比方：`lea` 给你**门牌号**，`mov` 从内存读则是**进屋把东西拿出来**。我们要告诉 `write`” 字符串在哪”，所以要的是门牌号——用 `lea`。

为什么用 RIP 相对，而不是写死地址

RIP 相对寻址让代码**不依赖自己被加载到哪个绝对地址**（位置无关）。现代系统普遍这么做。我们的翻译器必须正确处理它：在 JIT 里，因为翻译时就知道” 下一条指令地址”，所以这个目标地址其实是个**可以提前算死的常数**（见第 12 节）。

5.4 `movq $msglen, %rdx` ——第 3 个参数：长度

把字符串长度（汇编器算出来是 `0x28`，即十进制 40）放进 `rdx`，作为 `write` 的第 3 个参数 `count`（打印多少字节）。

我们真踩过的坑

最早这行用 Intel 语法写成 `mov rdx, msglen`，结果被汇编器理解成” 从地址 `0x28` 读内存”（编码成了 `48 8b 14 25 ...`，一条带 SIB 的内存加载），而不是” 把常数 40 放进去”。这是 Intel 语法里” 裸符号被当成内存操作数” 的经典陷阱。改成 AT&T 的 `$msglen`（`$` 强制立即数）后才正确。这也说明：**机器码骗不了人，必须反汇编核对**。

5.5 syscall ——真正请内核干活

万事俱备，执行 syscall。此刻寄存器状态是：rax=1（write），rdi=1（stdout），rsi= 字符串地址，rdx=40。内核据此把 40 个字节打印到屏幕，把” 实际写了多少字节” 放回 rax。

5.6 第二段：movq $60,%rax / xorq %rdi,%rdi / syscall

- movq $60，%rax：系统调用号 60 是 exit（退出程序）。
- xorq %rdi，%rdi：把 rdi 置 0，作为退出码（0 表示成功）。

为什么用 xor 把寄存器清零

xor（异或）有个性质：任何数和自己异或都等于 0。所以 xor %rdi，%rdi 就是把 rdi 清零。在 x86 上这比 mov $0，%rdi **编码更短**（3 字节 vs 7 字节），是编译器爱用的小技巧。

最后一条 syscall 触发 exit(0)，程序结束。整个程序就这么 8 条指令。

6 从汇编到机器码：逐字节拆解

把 hello.S 编译后反汇编，得到每条指令对应的真实字节：

Listing 2: objdump 反汇编（地址：字节：汇编）

```
401000: 48 c7 c0 01 00 00 00   mov rax, 0x1       ; write 调用号
401007: 48 c7 c7 01 00 00 00   mov rdi, 0x1       ; fd = stdout
40100e: 48 8d 35 eb 0f 00 00   lea rsi, [rip+0xfeb]; rsi = 0x402000 (msg)
401015: 48 c7 c2 28 00 00 00   mov rdx, 0x28      ; count = 40
40101c: 0f 05                  syscall
40101e: 48 c7 c0 3c 00 00 00   mov rax, 0x3c      ; exit 调用号 60
401025: 48 31 ff               xor rdi, rdi
401028: 0f 05                  syscall
```

x86 指令是**变长**的（这里从 2 字节到 7 字节不等），结构大致是：

[前缀] [操作码] [ModRM] [SIB] [位移] [立即数]

我们只需要看懂这几样。下面拆三条有代表性的。

6.1 拆 48 c7 c0 01 00 00 00 = mov rax，1

字节	角色	含义
0x48	REX 前缀	0100WRXB，这里 W=1 表示” 操作 64 位”
0xc7	操作码	C7 /0 = ” 把立即数搬进 r/m”
0xc0	ModRM	见下，指明目标是寄存器 rax
0x01 00 00 00	立即数	小端的 32 位数 = 1

ModRM 字节 0xc0 = 二进制 11 000 000，拆成三段：

- `mod = 11`：表示 r/m 是**一个寄存器**（不是内存）。
- `reg = 000`：这条指令里 reg 段被操作码当作扩展字段 `/0`，不是寄存器。
- `r/m = 000`：寄存器编号 0 = `rax`（结合 REX.B，这里仍是 0）。

REX 前缀是什么

x86 本来是 32 位的，扩展到 64 位时为了兼容，加了一个**可选的 1 字节前缀** REX（`0x40~0x4f`）。它的 4 个低位 `W R X B` 分别表示：W= 是否 64 位操作、R/X/B= 把寄存器编号扩展到能表示 `r8~r15`。`0x48` 就是只置了 W 位。

6.2 拆 `48 8d 35 eb 0f 00 00` = `lea rsi, [rip+0xfeb]`

字节	角色	含义
`0x48`	REX.W	64 位
`0x8d`	操作码	`8D` = `lea`
`0x35`	ModRM	`00 110 101`：见下
`0xeb 0f 00 00`	位移 disp32	小端 = `0x0feb` = 4075

ModRM `0x35` = `00 110 101`：

- `mod = 00` 且 `r/m = 101`：这是一个**特例**，表示 **RIP 相对寻址**，后面跟一个 32 位位移。
- `reg = 110`：寄存器编号 6 = `rsi`，是目标。

有效地址 = 下一条指令地址 + 位移 = `0x401015` + `0xfeb` = `0x402000`，正好是字符串 `msg` 的地址。`lea` 把这个地址放进 `rsi`。**解码器和翻译器都必须复现这套算法**。

6.3 拆 `0f 05` = `syscall`

最简单：两个字节的固定操作码。`0x0f` 是”双字节操作码”的引子，`0x0f 05` 整体就是 `syscall`。我们的解码器见到 `0x0f` 就再读一个字节，发现是 `0x05`，于是判定为系统调用。

7 rousaoe 总体架构

把上面的概念串起来，就是我们翻译器的全貌：

Listing 3: 数据流

```
x86-64 ELF 文件
     |
[加载器 loader.c]  把程序按地址装进内存，建好栈，rip 指向入口
     |
[解码器 decode.c]  从 rip 处读字节，还原成一条结构化指令 struct insn
     |
 +---+-------------------------------+
```

```
  |                                    |
[解释器 interp.c]                  [JIT  jit.c]
 逐条模拟执行                       把基本块翻成真正的 ARM64 机器码并原生执行
  |                                    |
  +---------------+--------------------+
                  |
        [系统调用层 syscall.c]  x86-64 的 syscall -> 宿主 ARM Linux 内核
```

两条后端二选一（默认解释器；设环境变量 `ROUSAOE_JIT=1` 走 JIT）：

- **解释器**：先保证**正确**。它就是把第 3 节那个” 取指-译码-执行” 循环用 C 写出来。
- **JIT**：再追求” 翻译成 ARM 二进制”。它把 x86 指令**真正编译成 ARM64 机器码**，放进可执行内存，让 ARM CPU 原生跑。这才是题目要的” 转译成 arm 的二进制”。

关键设计：地址直映射（identity mapping）

我们让 **guest（被翻译程序）的内存地址，直接等于宿主进程里的真实地址**。也就是说，程序里那个 `0x402000`，在我们进程里就真的放在 `0x402000`。好处是：guest 的指针不用任何转换，读写内存、把指针传给系统调用，都**直接用**。这是用户态翻译器常见的简化。能这么做，是因为我们的测试程序是**非 PIE 静态程序**，固定加载在低地址 `0x400000` 一带，这块地址在我们进程里正好空着。

7.1 虚拟 CPU：用一个结构体装下 x86 的状态

我们用一个 C 结构体表示”x86 处理器当前的样子”——16 个寄存器、指令指针、标志位：

Listing 4: src/cpu.h（节选）

```c
enum { RAX, RCX, RDX, RBX, RSP, RBP, RSI, RDI,
       R8,  R9,  R10, R11, R12, R13, R14, R15 };

struct x86cpu {
    uint64_t r[16];   /* 16 个通用寄存器，下标见上面的 enum */
    uint64_t rip;     /* 指令指针：下一条指令的地址 */
    uint64_t flags;   /* 标志位（零标志、符号标志等） */
    int      halted;  /* 程序是否已退出 */
    int      exit_code;
};
```

注意那个 `enum` 的顺序，正是第 4 节强调的 x86 寄存器编号顺序。后面所有部件都围着这个结构体转：解码器告诉我们” 这条指令动哪个寄存器”，解释器/JIT 就去改 `r[i]`。

8 第一步：ELF 加载器 loader.c

8.1 ELF 是什么

Linux 上的可执行文件、库，用的格式叫 **ELF**（Executable and Linkable Format）。一个 ELF 文件开头是**文件头**，告诉你” 这是不是 ELF、给哪种 CPU 的、入口在哪”；接着有一张 **程序头表**，每一项描述” 文件的哪一段要装到内存的哪个地址、可读还是可执行”。要装载的段类型叫 `PT_LOAD`。

8.2 加载器做三件事

1. **校验**：确认这是个给 x86-64 的、静态非 PIE 的 ELF（我们 MVP 只支持这种）。
2. **映射段**：把每个 `PT_LOAD` 段，用 `mmap` 放到它**指定的地址**（直映射），再把文件内容拷进去。
3. **建栈**：分配一块栈，按 Linux 约定在栈顶摆好 `argc/argv/环境变量/auxv`，让 `rsp` 指过去；最后把 `rip` 设成 ELF 入口地址。

核心代码：

Listing 5: src/loader.c（映射段，节选）

```
uint64_t va_start = ALIGN_DOWN(ph.p_vaddr, PAGE);
uint64_t va_end   = ALIGN_UP(ph.p_vaddr + ph.p_memsz, PAGE);
size_t   len      = va_end - va_start;

/* 在“段要求的地址”上申请内存（MAP_FIXED_NOREPLACE：地址被占就报错） */
void *m = mmap((void *)va_start, len, PROT_READ | PROT_WRITE,
               MAP_PRIVATE | MAP_ANONYMOUS | MAP_FIXED_NOREPLACE, -1, 0);

/* 把文件里这一段的内容拷到目标地址 */
pread(fd, (void *)ph.p_vaddr, ph.p_filesz, ph.p_offset);

/* 按段的权限（读/写/执行）重新设置保护位 */
mprotect((void *)va_start, len, prot_of(ph.p_flags));
```

为什么要页对齐

内存保护、映射都以**页**（page，通常 4096 字节）为最小单位。所以申请内存时要把起止地址对齐到页边界——这就是 `ALIGN_DOWN` / `ALIGN_UP` 在做的事。

装载完，`rip` = 入口 `0x401000`，`rsp` 指向我们建好的栈，舞台就绪。

9 第二步：指令解码器 decode.c

解码器把” 一串字节” 还原成” 一条结构化的指令”，存进 `struct insn`：它是哪种操作？动哪个寄存器？有没有内存操作数、地址怎么算？立即数是多少？

9.1 解码流程

对照第 6 节的指令结构，解码器顺序做：

1. 看开头是不是 REX 前缀（0x40~0x4f），是就记下 W/R/X/B 四位。
2. 读操作码。0x0f 开头要再读一字节（双字节操作码，如 syscall）。
3. 如果这条指令带 ModRM，就解析 mod/reg/r-m，必要时再读 SIB、位移。
4. 如果带立即数，按宽度读进来。

关键片段（处理带 ModRM 的指令，含 RIP 相对这个特例）：

Listing 6: src/decode.c（ModRM 解析，节选）

```
uint8_t modrm = *c++;
int mod = modrm >> 6;                  /* 高 2 位 */
int reg = ((modrm >> 3) & 7) | (rex_r << 3);
int rm  = (modrm & 7)       | (rex_b << 3);

if (mod == 3) {                        /* r/m 是一个寄存器 */
    ins->rm_is_mem = 0;
    ins->rm_reg = rm;
} else {                               /* r/m 是内存 */
    ins->rm_is_mem = 1;
    int rm3 = modrm & 7;
    if (rm3 == 5 && mod == 0) {        /* 特例：RIP 相对寻址 */
        ins->rip_rel = 1;
        ins->disp = rd32(&c);          /* 读 32 位位移 */
    }
    /* ... 其他寻址方式 ... */
}
```

我们的 MVP 只需要认得 4 类指令，所以解码器只覆盖这几个操作码：mov r,imm（0xB8+r 和 0xC7 /0）、lea（0x8D）、xor r/m,r（0x31）、syscall（0x0F 05）。结构虽小，但 REX / ModRM / SIB / RIP 相对这套**骨架**是完整的，以后加指令只是往 switch 里添分支。

10 第三步：解释器 interp.c

解释器就是把” 取指-译码-执行” 循环写成 C。一个大 for 循环：解码出一条指令，按它的种类去改虚拟 CPU 的状态，挪动 rip，再来一遍，直到程序退出。

Listing 7: src/interp.c（主循环，节选）

```
for (;;) {
    struct insn ins;
    int n = x86_decode((const uint8_t *)cpu->rip, &ins);  /* 取指+译码 */
    uint64_t next = cpu->rip + n;                          /* 下一条地址 */
```

```
5
6       switch (ins.op) {                                   /* 执行 */
7       case OP_MOV_RI:                         /* 把立即数写进寄存器 */
8           cpu->r[ins.reg] = ins.rex_w ? (uint64_t)ins.imm : (uint32_t)ins.imm;
9           break;
10      case OP_LEA:                            /* 算出地址放进寄存器（不读内存） */
11          cpu->r[ins.reg] = eff_addr(cpu, &ins, next);
12          break;
13      case OP_XOR_RM_R: {                     /* r/m = r/m XOR reg */
14          uint64_t res = cpu->r[ins.rm_reg] ^ cpu->r[ins.reg];
15          cpu->r[ins.rm_reg] = res;
16          set_logic_flags(cpu, res, ins.rex_w);
17          break;
18      }
19      case OP_SYSCALL:                        /* 交给系统调用层 */
20          cpu->rip = next;
21          handle_syscall(cpu);
22          if (cpu->halted) return cpu->exit_code;
23          continue;
24      }
25      cpu->rip = next;
26  }
```

逐条对照第 5 节就能看懂：`OP_MOV_RI` 把立即数放进 `cpu->r[ins.reg]`；`OP_LEA` 调用 `eff_addr` 算出 RIP 相对地址；`OP_XOR_RM_R` 做异或并更新标志位；遇到 `OP_SYSCALL` 就转交系统调用层。

“有效地址”怎么算

`eff_addr` 复现了第 6 节那条公式：RIP 相对时，地址 = 下一条指令地址 + 位移。因为我们用了地址直映射，这个地址可以**直接当宿主指针用**。

跑一下，解释器能正确打印出那行字。但请注意：解释器**从头到尾没有生成任何一条 ARM 指令**，它只是”用 C 模拟 x86”。真正”翻译成 ARM 二进制”是下一节的事。

11 第四步：系统调用转译 syscall.c

程序执行 `syscall` 时，我们要把这个”x86-64 Linux 的系统调用请求”，变成宿主”ARM64 Linux”上真实发生的事。

两边的系统调用号不一样

同一个服务，在 x86-64 和 ARM64 上**编号不同**：例如 `write` 在 x86-64 是 1，在 ARM64（AArch64）是 64；`read` 在 x86-64 是 0，ARM64 是 63。传参寄存器也不同（x86-64 用 `rdi/rsi/...`，ARM64 用 `x0/x1/...`）。所以中间必须有一层”翻译”。

我们的做法很务实：不直接发原始系统调用，而是**按 x86-64 的号去调用宿主的 C 库函数**，

由 C 库在 ARM64 上发出正确的本地系统调用。因为指针是直映射的，`write` 要的缓冲区地址可以直接用。号码的转换就**隐含**在这个 `switch` 里：

Listing 8: src/syscall.c（节选）

```c
switch (nr) {                          /* nr = rax = x86-64 系统调用号 */
case 1: {                              /* write(fd, buf, count) */
    ssize_t ret = write((int)a1, (const void *)a2, (size_t)a3);
    cpu->r[RAX] = (ret < 0) ? -errno : ret;   /* 返回值写回 rax */
    break;
}
case 60:                               /* exit */
case 231:                              /* exit_group */
    cpu->halted = 1;
    cpu->exit_code = (int)a1;
    break;
}
```

其中 `a1/a2/a3` 就是从 `rdi/rsi/rdx` 取出的参数。`write` 的返回值按 Linux 约定写回 `rax`。`exit` 则把虚拟 CPU 标记为已退出。

12 第五步：JIT ——真正翻译成 ARM 机器码

这是全项目的核心，也是题目字面要求的” 把 x86 转译成 arm 的二进制”。解释器是” 用 C 模拟”，JIT 则是**当场生成真实的 ARM64 机器码字节，写进一块可执行内存，然后让 ARM CPU 直接跑**。

12.1 基本思路：以” 基本块” 为单位翻译

基本块（basic block）= 一段顺直执行、中间不分叉的指令。我们的派发器（dispatcher）这样转：

1. 看 `rip` 处的基本块翻译过没有（查缓存）。
2. 没翻过就翻：从 `rip` 逐条解码，把每条 x86 指令**发射**成等价的 ARM64 指令，直到遇到 `syscall` 就结束这个块。
3. 调用这块生成的 ARM64 代码（它会更新虚拟 CPU 的寄存器）。
4. 回到 C，处理那个 `syscall`，然后继续下一个块。

12.2 寄存器怎么映射

最简单可靠的办法：guest 的 16 个寄存器仍然**存在内存里的 `cpu->r[]` 数组**。生成的 ARM64 代码约定：

- `x0` 始终存放**指向 `struct x86cpu` 的指针**（按 ARM 调用约定，第 1 个参数就在 `x0`）。

- x9、x10 当临时寄存器。

于是”读 guest 寄存器”就是从 [x0 + 偏移] 用 ldr 载入，”写回”就是 str。

12.3 我们要会发射的几条 ARM64 指令

ARM64 指令是**定长 4 字节**（比 x86 规整多了）。我们只需要会”手写”这几条的机器码：

ARM64 指令	作用	我们用它来
movz / movk	把 16 位一段一段拼出常数	构造立即数、地址
str Xt,[Xn,#off]	把寄存器写进内存	写回 guest 寄存器
ldr Xt,[Xn,#off]	从内存读进寄存器	读取 guest 寄存器
eor Xd,Xn,Xm	异或	实现 x86 的 xor
ret	返回	块结束，回到派发器

每条指令的 4 个字节怎么拼，是查 ARM 手册得到的固定公式。例如 movz：

Listing 9: src/jit.c（ARM64 指令编码器，节选）

```c
/* MOVZ Xd, #imm, LSL #shift —— 把 16 位立即数放到某一段 */
static uint32_t a_movz(int rd, uint16_t imm, int shift)
{ return 0xD2800000u | ((shift/16) << 21) | ((uint32_t)imm << 5) | rd; }

/* STR Xt, [Xn, #imm12*8] —— 写内存 */
static uint32_t a_str(int rt, int rn, int imm12)
{ return 0xF9000000u | ((uint32_t)imm12 << 10) | (rn << 5) | rt; }

/* EOR Xd, Xn, Xm —— 异或 */
static uint32_t a_eor(int rd, int rn, int rm)
{ return 0xCA000000u | (rm << 16) | (rn << 5) | rd; }
```

一个 64 位常数要拆成几段

ARM64 一条指令塞不下 64 位立即数，所以用 movz 放最低 16 位，再用最多 3 条 movk 逐段补上高位。比如地址 0x402000 会生成两条：movz x9,#0x2000 和 movk x9,#0x40,lsl#16，拼出 0x402000。

12.4 翻译循环

把上面这些编码器串起来，就是翻译一个块：

Listing 10: src/jit.c（翻译一个基本块，节选）

```c
switch (ins.op) {
case OP_MOV_RI:                 /* mov reg, imm */
    emit_imm64(&p, T0, imm);              /* 把常数装进 x9 */
    *p++ = a_str(T0, CPU, UNIT_R(ins.reg)); /* 写回 cpu->r[reg] */
```

```
    break;
case OP_LEA:                    /* lea reg, [rip+disp]：地址是常数 */
    emit_imm64(&p, T0, next + ins.disp);
    *p++ = a_str(T0, CPU, UNIT_R(ins.reg));
    break;
case OP_XOR_RM_R:               /* r/m = r/m XOR reg */
    *p++ = a_ldr(T0, CPU, UNIT_R(ins.rm_reg));  /* x9  = cpu->r[rm] */
    *p++ = a_ldr(T1, CPU, UNIT_R(ins.reg));     /* x10 = cpu->r[reg] */
    *p++ = a_eor(T0, T0, T1);                   /* x9 = x9 ^ x10 */
    *p++ = a_str(T0, CPU, UNIT_R(ins.rm_reg));  /* 写回 */
    break;
}
/* 遇到 syscall：把块尾地址写进 cpu->rip，再 ret 回派发器 */
```

别忘了刷新指令缓存

我们刚把机器码字节**当数据**写进内存，但 CPU 取指走的是**指令缓存**。必须调用 `__builtin___clear_cache` 让数据缓存和指令缓存同步，否则 CPU 可能执行到旧内容。这是自修改/即时生成代码的必备步骤。

12.5 看真实生成的 ARM64

下面是 JIT 为 hello 的**第一个基本块**真正生成的 ARM64（我们把生成的字节 dump 出来反汇编）：

Listing 11: x86-64 基本块 -> 生成的 ARM64（真实输出）

```
d2800029  mov  x9, #0x1             ; mov rax,1
f9000009  str  x9, [x0]             ;   -> cpu->r[RAX]   (偏移 0)
d2800029  mov  x9, #0x1             ; mov rdi,1
f9001c09  str  x9, [x0, #56]        ;   -> cpu->r[RDI]   (偏移 7*8=56)
d2840009  mov  x9, #0x2000          ; lea rsi, msg
f2a00809  movk x9, #0x40, lsl #16 ;   x9 = 0x402000
f9001809  str  x9, [x0, #48]        ;   -> cpu->r[RSI]   (偏移 6*8=48)
d2800509  mov  x9, #0x28            ; mov rdx,0x28 (=40)
f9000809  str  x9, [x0, #16]        ;   -> cpu->r[RDX]   (偏移 2*8=16)
d2820389  mov  x9, #0x101c          ; 把 syscall 的地址...
f2a00809  movk x9, #0x40, lsl #16 ;   ...0x40101c
f9004009  str  x9, [x0, #128]       ;   -> cpu->rip      (偏移 128)
d65f03c0  ret                       ; 回到派发器处理 syscall
```

逐行对照第 5 节的 x86 程序，你会发现**完全一一对应**：每条 x86 `mov` 都变成了”装常数 + 写回内存”两条 ARM 指令；`lea` 算出的地址 `0x402000` 正是字符串地址；偏移 `0/56/48/16` 正好是 `rax/rdi/rsi/rdx` 在数组里的位置。这串 ARM64 字节在 ARM CPU 上**原生执行**，更新好寄存器后返回，派发器再发出 `write`。**这就是”把 x86 翻译成了 ARM 二进制”**。

13 把它跑起来：容器、构建与验证

整个项目在一个 **linux/arm64 的 Docker 容器**里构建运行（容器内是原生 ARM64 Linux），用 x86-64 交叉编译器产出测试程序，再用我们的翻译器去跑它。一条命令搞定：

Listing 12: 构建并验证

```
make docker-test
```

它会：编译翻译器 `rousaoe`（ARM64 原生）；交叉编译 `tests/hello.S` 成 x86-64 程序；先用**解释器**跑、再用 **JIT** 跑、最后反汇编展示生成的 ARM64。真实输出：

Listing 13: 运行结果

```
== interpreter (host: aarch64) ==
hello from x86-64, translated to arm64!
== JIT: translate x86-64 -> ARM64, run natively ==
hello from x86-64, translated to arm64!
```

一个为 **x86-64** 编译的程序，在 **ARM64** 机器上，通过我们自己写的翻译器，打印出了正确结果——两条后端都通过。

为什么本文档也用容器渲染

连这份 PDF 也是在一个 linux/arm64 容器里用 **XeLaTeX + ctex + Fandol 中文字体**排版出来的（见 `doc/latex.Dockerfile`），和项目"一切在容器里可复现"的风格一致。

14 现状边界与下一步

诚实地说，这是一个**证明思路可行的最小翻译器**，还不能跑任意真实程序。

能力	现状	说明
加载静态 x86-64 ELF	✓	非 PIE、地址直映射
指令集	部分	仅 `mov/lea/xor/syscall`
系统调用	部分	仅 `write/exit`
解释器后端	✓	正确优先
JIT 生成 ARM64	✓	基本块级，真实原生执行
分支 / 控制流	✗	块只到 syscall 为止
浮点 / SSE、多线程、信号	✗	未实现
动态链接（glibc 程序）	✗	需要更全的栈与系统调用

下一步（按优先级）：

1. **扩指令 + 控制流**：加 `add/sub/cmp/jmp/jcc/call/ret/push/pop`，让基本块能跨分支——这是跑任何真实程序的前提。
2. 跑通一个真实的 `glibc` 静态程序（补全 auxv 与一批系统调用）。

3. 性能：块链接、间接跳转缓存、把热点寄存器固定映射到 ARM 寄存器、JIT 里做惰性标志位。

4. SSE→NEON、多线程与内存序（x86 强序 vs ARM 弱序）、信号、自修改代码。

附录 A：术语表

ISA（指令集架构）

CPU 与软件的”合同”：有哪些寄存器、哪些指令、怎么编码。

寄存器

CPU 内部数量很少但极快的存储位置。x86-64 有 16 个 64 位通用寄存器。

立即数（immediate）

直接写在指令里的常数，如 `mov $1` 里的 1。

机器码 / 汇编

前者是 CPU 吃的二进制字节；后者是它的人类可读别名，一一对应。

ModRM

x86 指令里描述”操作数是寄存器还是内存、是哪个”的一个字节。

REX 前缀

x86-64 为支持 64 位与扩展寄存器加的可选 1 字节前缀。

系统调用（syscall）

程序请操作系统内核干活（打印、退出等）的机制。

ELF

Linux 的可执行文件格式。

DBT（动态二进制翻译）

程序边跑边把指令翻成目标 CPU 的机器码并缓存。

JIT（即时编译）

运行时生成机器码并立即执行——本项目”翻译成 ARM 二进制”的部件。

基本块

一段顺直、中间不分叉的指令，翻译的最小单位。

地址直映射

让 guest 的内存地址直接等于宿主进程里的真实地址，省去指针转换。

附录 B：本项目用到的指令速查

x86-64	机器码	作用
`mov reg, imm`	`0xC7 /0` 或 `0xB8+r`	把常数放进寄存器
`lea reg, [rip+d]`	`0x8D`	算出地址放进寄存器（不读内存）
`xor r/m, reg`	`0x31`	异或；与自身异或可清零
`syscall`	`0x0F 05`	请内核服务
ARM64	**机器码前缀**	**作用**
`movz/movk`	`0xD28../0xF2A..`	分段拼出立即数
`ldr/str`	`0xF940../0xF900..`	读 / 写内存
`eor`	`0xCA0..`	异或
`ret`	`0xD65F03C0`	返回