

纯浏览器内 ARM 指令集架构模拟及模拟环境下部分 C/C++标准库的简易实现

上海海洋大学 雷适嘉<me@ryubai.com>

摘要:

随着互联网技术的发展,以往专门针对目标系统开发桌面客户端程序的方法越来越多的被开发通用的浏览器前端程序所替代。然而,一般前端开发方式所常用的 JavaScript 语言不仅运行效率较低,其对于有精确类型和结构的应用场景也缺少支持。本文讨论利用最新的浏览器中支持的 WebAssembly 技术,将 ARMv4T 指令集模拟器开发并编译至浏览器内运行的同时,围绕该指令集模拟器实现一个简单的类嵌入式平台 wasm_arm,并通过前端常用的 HTML + JavaScript + CSS 实现其可视化。本文也利用了该平台研究并验证了 ARM 处理器的 C++二进制接口,并讨论了将一部分重要的 C++基础库功能在该类嵌入式平台上的实现方式。此外,本文亦讨论了一种利用 GBA 游戏机内置的 ARM7TDMI 处理器针对该指令集模拟器进行自动化执行的模拟准确性验证方案。

关键词: ARM ISA, C/C++ ABI, C++, WebAssembly, HTML, GBA.

An In-browser ARM Instruction Set Architecture Simulator with Certain C++ Standard Library Implementation

Lei Shijia <me@ryubai.com>
Shanghai Ocean University

Feb 26, 2021

Abstract

Following the advancement of modern-day Internet technology, the former generic methodology of developing desktop client applications targeting a specific operating system is gradually becoming obsolete and usually replaced by a universal front-end application executing inside browsers. However, the JavaScript language commonly used for front-end application development is inefficient and lacks support for application scenarios containing precise data type and structure usage. This paper discusses a practical way to develop and compile an in-browser ARMv4T Instruction Set simulator with a minimalized embedded environment using WebAssembly technology and implements its visualization with generic front-end HTML5. This paper also studies and verifies the C++ application binary interface of the ARM architecture and implements certain C++ memory management routines for the said simulation environment. Furthermore, this paper discusses a verification method on execution accuracy for an ARM processor simulator utilizing the ARM7TDMI processor within a Gameboy Advance handheld game console.

CONTENTS

1	Introduction.....	1
2	Existing Work	3
2.1	WebAssembly.....	3
2.2	Emscripten Compiler.....	3
2.3	Visualization.....	4
2.4	Arm Instruction Set Architecture	4
2.5	Simulation Verification and Debugging.....	4
3	ARM Instruction Set Architecture(ISA) Simulation	6
3.1	Comparing Instruction Decode and Execution Flow in ARMv4 Thumb/ARM ISA with MIPS ISA.....	6
3.2	Comparing Stack Machine in ARM ISA with X86_64 ISA	7
3.3	Pipelined Bus Accessing Modeling and Cycle Timing Correctness in ARM ISA.....	11
3.4	A C++ Implementation.....	14
3.5	Issues Encountered During Implementation.....	17
3.5.1	Cross-boundary Bus Accessing and Memory Alignment.....	17
3.5.2	Block Data Transfer Sub-stages.....	19
3.5.3	The Importance of Emulating Pipeline Faithfully.....	20
3.6	A General Approach on Semi-automatic ARM ISA Emulator Debugging	21
3.6.1	Modeling ARM Processor with Finite State Machine	21
3.6.2	Semi-automatic Semi-random Test-case Generation.....	22
3.6.3	Automatic Evaluation with Physical Machine	23
3.7	ARM Emulation Performance Optimization.....	27
3.7.1	Instruction Pre-decoding and Condition Code Pre-generation at Compile Time	27
3.7.2	Comparing Reinterpret Cast with Bit Shift.....	29
4	A Minimal ARM Micro-architecture Inside Web Browser.....	31
4.1	Basic Design.....	31
4.1.1	Memory Model, Bootloader, and Booting Sequence.....	31
4.1.2	Bridging with the WebAssembly	33
4.2	ARM C/C++ Application Binary Interface Verification.....	34
4.2.1	Function Routine and Variable Lifetime.....	34
4.2.2	From Object-oriented Programming to Binary Instructions	37
4.3	Certain C/C++ Standard Library Implementation for WASM_ARM Micro- architecture	43
4.3.1	Memory Management	43
4.3.2	Smart Pointers	45
4.3.3	Output Interface.....	48
4.4	An ARM ISA Emulator inside Itself.....	49
5	Conclusion	50

目录

1 引言.....	1
2 已有工作.....	3
2.1 WebAssembly 技术简介	3
2.2 Emscripten 编译链简介	3
2.3 利用 WEB 前端进行模拟可视化的实现.....	4
2.4 ARM 指令集架构及其常见模拟器简介	4
2.5 ARM ISA 模拟的时钟及执行的准确性验证与 GBA 游戏机简介.....	4
3 ARM 指令集架构模拟，测试及优化	6
3.1 ARMv4 的 arm、thumb 指令集架构与 MIPS 架构的指令译码及执行流程对比.....	6
3.2 对比 ARM 架构与 X86_64 架构的栈机器.....	7
3.3 ARM 架构的总线的流水线访问模拟及其时钟准确性设计.....	11
3.4 一个 C++的实现方案	14
3.5 实现中遇到的问题.....	17
3.5.1 跨边界地址访问和内存对齐.....	17
3.5.2 块数据传输指令的子阶段.....	19
3.5.3 正确模拟流水线的重要性.....	20
3.6 一个半自动化的 ARM 指令集架构模拟器的测试方法	21
3.6.1 ARM 处理器的有限状态机建模	21
3.6.2 半自动化半随机测试集生成.....	22
3.6.3 自动化实机交叉检验.....	23
3.7 ARM 模拟器效率优化	27
3.7.1 指令编译期预解码和状态码编译期预生成.....	27
3.7.2 对比 reinterpret_cast 和 Bit Shift	29
4 浏览器内的简单 ARM 微架构 WASM_ARM	31
4.1 基本设计.....	31
4.1.1 内存模型，Bootloader，以及启动初始化流程.....	31
4.1.2 连接 WebAssembly	33
4.2 验证 ARM 的 C/C++二进制程序接口	34
4.2.1 函数调用过程及变量生命周期.....	34
4.2.2 从面向对象到处理器二进制指令.....	37
4.3 WASM_ARM 微架构上简化的 C/C++标准库部分功能实现	43
4.3.1 内存管理.....	43
4.3.2 智能指针.....	45
4.3.3 输出接口.....	48
4.4 ARM 指令集架构模拟内的 ARM 指令集架构模拟.....	49
5 总结.....	50
附录.....	51
参考文献.....	52

1 引言

网页中也能进行密集型计算吗？

现代的前端程序作为某种含义上的“窗口”，其作用一般是提供给用户一个人机交互的接口，即，将用户的输入转换为后端可识别的数据和将后端传出的数据转换为用户可理解的文字或画面。前端一般不涉及密集计算，其计算内容也主要用于类似于画面特效的实现（如利用 WebGL），或数据的解析写入 DOM 与持久化（如暂存 cookie）。

前端开发从语言层面上也一般指普通浏览器默认支持的 Javascript 语言。然而，该语言的自身特性，如需要运行期解释开销（尽管可采用 JIT 优化），不支持精确类型等，限制了该语言自身可以应用的使用场景。

例如在早些时候，我曾利用 Javascript 语言实现过针对任天堂 NES 游戏机的模拟，由于 NES 的 6502 处理器为 8 位，因此需要对大部分数值采用类似如下的策略进行精度限制：

```
this.x &= 0xFF;
```

此外，由于在 Javascript 中没有用户显式定义的有符号/无符号类型变量的区分，如下的两行代码：

```
console.log(0xFFFFFFFF)
console.log(0xFFFFFFFF & 0xFFFFFFFF)
```

两行的输出竟因位运算符&的存在变得完全不同：

```
4294967295
-1
```

诚然，二者都代表同一二进制数字，输出结果都是没有问题的，但这种没有存在必要的“隐式”类型转换本身又意味着运行期开销。总之，就算抛开执行效率不谈，关于精确数据处理方面的逻辑实现上 Javascript 也并非一门适合的语言，毕竟为“抵御”其语法特性本身做出开销实在是一件不可理喻的事情。

随着时代的发展，越来越多的桌面程序开始转换为 Web 前端程序，以降低专门针对对应操作系统分别开发应用程序的开销。而其中计算较为密集的程序的前端实现就变得极为困难，Javascript 语言动态、简单的优势变成了劣势。

而 WebAssembly 的出现，使得在浏览器内运行计算密集型处理逻辑成为可能，该技术通过一个较高效率的虚拟机（类似 JVM）执行 wat 字节码（类似 Java 字节码）在浏览器内运行，其能达到比原生 Javascript 更快的执行速度。同时，该虚拟机较为简单，如其并没有内存管理机制，其编译过程也仅作为 llvm 的后端实现，从设计上保障了更高的执行效率。并且在目标开发语言上可选用 C++、GO 等更适合用于计算、且能被 llvm 前端解析的语言。

本文的内容则是利用 WebAssembly 的这一优势，讨论一个完全运行于浏览器 WebAssembly 运行环境内的小端序 ARMv4T 指令集（包含 arm 指令集及 thumb 指令集）架构（下文简称 arm 架构）模拟平台（下文简称 wasm_arm 平台）、对该平台的 C++实现、对该平台的模拟正确性的验证方案、实现兼容该平台的部分 C++基础库函数（仅追求实现部分基础的功能）、以及利用该库在该模拟平台内编译执行模拟器自身。

- a. 本文的核心部分将主要包含以下重点内容：
1. 在浏览器中运行的，时钟准确且执行准确的、快速的、尽量独立（即不依赖其它库）的、高度可视化的 ARMv4T 指令集架构模拟器设计及开发方案及实现。
 2. 利用任天堂 GBA 游戏机对该模拟器的执行结果进行准确性验证的方案。
 3. 利用该模拟实现的简易类嵌入式运行环境 `wasm_arm`。
 4. 实现一个在该简易类嵌入式运行环境中执行的 `bootloader`。
 5. 实现该简易类嵌入式环境下的一部分 C/C++ 标准库的功能。
 6. 利用该环境研究并验证 C++ 的二进制程序接口。
- b. 得益于 WEB 前端自身的特点配合 WebAssembly 的优势，该简易类嵌入式运行环境 `wasm_arm` 将包含以下特性：
1. 无需独立客户端，页面加载完毕即运行。
 2. 运行快速，准确（相较于传统的浏览器内的 JavaScript 而言）。
 3. 无后端运行期负载，后端仅需如同提供静态页面一样提供编译后的 WASM 二进制程序给前端即可。前端 JavaScript 负责初始化 WebAssembly 运行环境、并与此环境中程序交互以实现可视化。
- c. 本文的章节安排：
- 本文的第二章介绍了现有的技术，包含文中所用到的 WebAssembly 及 GBA 游戏机等，同时也简要介绍了目前已有的 ARM 指令集模拟器。
- 本文的第三章和第四章为正文部分，其主要围绕两大方向展开：一是实现一个 ARM 指令集模拟器，该内容集中在第三章，并且包含了对它的开发，测试，优化的全部过程；二是利用该模拟器实现一最小嵌入式环境的同时利用该环境验证、讨论及实现 C++ 的一部分库功能，该内容集中在第四章，其中 4.1 讨论了该极简嵌入式环境 `wasm_arm` 的实现，剩余部分利用 `wasm_arm` 环境并从底层角度入手，验证讨论或实现了 C++ 的函数调用、面向对象、内存管理、智能指针的部分。
- 本文通过实现浏览器前端内的高度可视化 ARM 模拟器这一示例，讨论了一种在前端实现对具有数据结构和类型准确性要求的应用程序的开发方式。展示了 WebAssembly 技术所带来的对于未来的前端应用开发的新方向。

本文所描述的 ARM 架构模拟平台 `wasm_arm`：

http://ryubai.com/e/wasm_arm

针对该平台编译的 C++ 示例及部分 C++ 标准库功能实现：

https://github.com/toshirodesu/wasm_arm-devkit

利用任天堂 GBA 游戏机的 arm 处理器核心对 arm 指令集架构模拟器进行测试：

https://github.com/toshirodesu/gba_armv4t_instruction_verification

完全由 JavaScript 编写的浏览器内 NES (FC) 任天堂红白机模拟器：

<http://ryubai.com/nejs>

<https://github.com/toshirodesu/nejs>

此外：

1. 下文所有的 x86_64 汇编代码格式均采用 AT&T syntax。
2. 编译参数如段落中未具体说明请参见附录。

2 已有工作

2.1 WebAssembly 技术简介

WebAssembly 是一个新兴的被浏览器广泛支持的栈虚拟机，支持 C、C++、GO 等多种语言，并有着运行快速（相对浏览器内 JavaScript）、用户体验舒适（无需安装过程）、支持广泛（四大浏览器的新版本均支持，包含 chrome、firefox、safari 和 edge）的优点。

WebAssembly 技术简单来说主要由两套标准组成：第一是一套浏览器中运行的栈机器的 wasm 二进制字节码标准。第二是 wasm 二进制字节码标准所对应的，可称为 WebAssembly 机器码的汇编语言的 WebAssembly 文本格式（WebAssembly Text Format, WAT）标准，其语法类似 lisp。WebAssembly 可以接近原生的性能运行，并为诸如 C/C++ 等语言提供一个编译目标，以便它们可以在 Web 上运行^[1]。WebAssembly 也被设计为可以与 JavaScript 共存，允许两者一起工作。开发者可以使用 C/C++ 代码，使用 Emscripten 这样的编译器工具将其翻译成 WASM，并将生成的 WASM 模块加载到 JavaScript 应用程序中，在那里它将被浏览器安全有效地执行。

WebAssembly 是作为 W3C WebAssembly 社区组内的开放标准创建的，目标如下：

1. 快速、高效和可移植—通过利用通用硬件功能，可以在不同平台上以接近本机的速度执行 WebAssembly 代码。
2. 可读性和可调试性。WebAssembly 是一种低级汇编语言，但它也有一种人类可读的文本格式（WAT），允许手工编写、查看和调试代码。
3. 保持安全。WebAssembly 被指定在安全的沙盒执行环境中运行。与其他 web 代码一样，它将强制执行浏览器的同源和权限策略。
4. 不要破坏 web。WebAssembly 的设计使其能够很好地与其他 web 技术配合，并保持向后兼容性。

WebAssembly 是一个新兴的标准，其制定者们也正在进行规范方面的工作。浏览器厂商已经就最初的 wasmapi 和二进制格式的设计达成了共识，并且有一个活跃的 W3C 社区小组，成员来自 Mozilla、微软、谷歌和苹果。

2.2 Emscripten 编译链简介

针对 wasm 二进制格式可采用 emscripten 编译器，该编译器利用了 llvm 及其 clang 前端，实现了由 C/C++ 等语言编译至 wasm 机器码的功能^[2]。利用 emscripten 编译链配合 c++ 语言开发的在浏览器中执行的 wasm 二进制程序不仅相较于传统的 JavaScript 语言可以有效提升运行速度，同时可以利用其语言特性组织健壮的运行逻辑。Emscripten 是 WebAssembly 的一个完整的开源编译器工具链。

Emscripten 对 C/C++ 代码的支持是相当全面的。其支持 C 标准库、C++ 标准库、C++ 异常等，甚至包含 SDL2 等常用 API。

当然，本机和 Emscripten 运行时环境之间存在差异，这意味着在移植的情况下通常需对本机代码进行一些更改。许多应用程序一般只需要改变它们定义主循环的方式，如果涉及到文件访问还需要修改它们的文件处理逻辑以适应浏览器/JavaScript 的限制。

2.3 利用 WEB 前端进行模拟可视化的实现

由于 wasm 的运行环境独立在沙箱中进行，UI 的更新则可以同传统前端程序一样，继续使用 JavaScript 操作 DOM。JavaScript 的运行环境同 WebAssembly 运行环境二者间采用函数绑定和专门开辟位于 WebAssembly VM 内的堆内存区域共享的方式进行交互。二者各自封装，互不干扰。

2.4 ARM 指令集架构及其常见模拟器简介

本文将以 ARM7TDMI 型号处理器为模拟对象，针对 ARMv4T 指令集进行包含总线访问时钟延迟特性的指令集功能性模拟。目前常见的针对该架构的模拟器有：

Fixed Virtual Platforms (FVPs)：

<https://developer.arm.com/tools-and-software/simulation-models/fixed-virtual-platforms>

arm 公司官方开发的闭源跨平台模拟器 FVPs。

其对处理器新型号的支持速度比实体处理器往往更加提前，使得开发者可以在还未取得新型处理器的实体硬件时便提前在其上开发。

FVPs 所代表的完整 arm 系统模型也不仅仅是一个 arm 指令集模拟器。通过在 FVPs 中建模的处理器、内存和其他外围设备，可以很好地模拟软件在物理设备上的执行方式。

FVPs 亦很容易扩展，易于消除软件开发和验证对硬件设备的依赖。这在为 DynamIQ、Helium、SVE 等新技术开发代码时尤其有用，因为访问其对应硬件平台的机会非常有限。

Crossware ARM Simulator：

<https://www.crossware.com/arm/simulator>

Crossware 的 arm 模拟器，其特点是包含一个针对调试器的公共接口。

VisUAL：

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

VisUAL 是一个跨平台的 arm 架构汇编语言学习工具。

它除了模拟了 arm ual 指令集的一个子集外，还提供了一些汇编语言编程特有的关键概念的可视化，因此有助于使 arm 汇编编程更易于学习和使用。

它是专为伦敦帝国理工学院电气与电子工程系计算机系的结构入门课设计的教学工具。

CPULATOR：

<https://cpulator.01xz.net/?sys=arm-delsoc>

CPULATOR 是一个在 web 浏览器中运行的计算机系统（包含处理器、内存和 I/O 设备）的模拟器和调试器。使用该模拟器可免除使用硬件的需要运行和调试程序。该模拟系统基于 Altera 联合实验室计划（NIOSII 和 ARMv7）和 SPIM（MIPS）的计算机系统。

2.5 ARM ISA 模拟的时钟及执行的准确性验证与 GBA 游戏机简介

Game Boy Advance (GBA) 是由任天堂开发、制造和销售的 32 位掌上游戏机。该游戏机采用 ARM7TDMI 作为处理器核心，搭配任天堂自行设计的图像处理电路，封装到一块 AGB CPU 芯片内。

本文参考了对 GBA 游戏机展开研究的部分相关文章的内容，对任天堂如何利用 arm 处理器的边界特性探测异常环境在相关章节进行了引用及说明，并在开发的模拟器中准确实现这些特性。

本文利用了实体 GBA 机器和对针对其处理器模拟的指令执行的结果准确性进行了验证。将模拟器编译至 GBA 游戏机中，使得 GBA 实机和模拟器同时在 GBA 内执行相同的指令，并将 GBA 实机的运行结果和模拟器内执行产生的结果进行比对。本部分利用了一种半自动的处理器模拟器测试方法：通过半随机生成的测试用例，以验证处理器是否被正确模拟。通过在模拟的处理器和物理处理器上同时运行相同的测试用例，并在执行后比较两者的状态，可以检测到模拟器的不正确行为。并利用最终状态的差异查找出模拟器代码中的缺陷。

3 ARM 指令集架构模拟，测试及优化

3.1 ARMv4 的 arm、thumb 指令集架构与 MIPS 架构的指令译码及执行流程对比

ARMv4 架构采用三段指令流水线（分别为 Fetch，Decode，Execute），指令参数被包裹进指令内，指令全部定长。其包含有 32 位的 arm 指令集，和紧凑的 16 位 thumb 指令集^[3]。r15 寄存器作为程序计数器（Program Counter）永远指向最后进入流水线的指令（因此计算跳转地址需要额外减掉两个指令的宽度）。

由此可知流水线中各指令的实际内存地址为：

	arm 指令集	thumb 指令集
1. [Fetch]	r15	r15
2. [Decode]	r15 - 4	r15 - 2
3. [Execute]	r15 - 8	r15 - 4

表 3-1 流水线中各指令的实际内存地址

例如，下方为 ARMv4 架构下顺序执行中、使用 arm 指令集、小端序配置、r15 寄存器值为 0x00001008 时的流水线。此刻正在执行 0x00001000 处指令，即 0x03020100。

			r15 指向↓	
00001000h	+0 +1 +2 +3	+4 +5 +6 +7	+8 +9 +a +b	+c +d +e +f
Pipeline Stage	[Execute]	[Decode]	[Fetch]	
Data	0x03020100	0x07060504	0x0b0a0908	0x0f0e0d0c

表 3-2 流水线执行状态示例

对比 MIPS 架构的指令流水线，此处拿 IDT R3000 处理器举例^[4]。

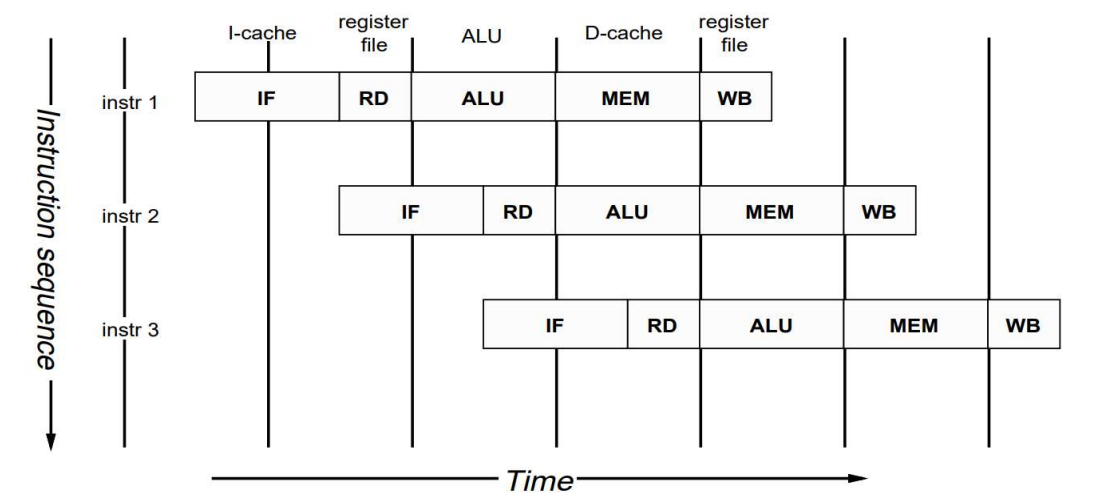


图 3-1 IDT R3000 处理器指令流水线

可以看到，IDT R3000 处理器由于将单个指令内部执行阶段分解为包含有缓存读写的不同子阶段并暴露给程序员，程序员必须手动或利用编译器自动进行指令重排避免指令流水线读写时刻差产生的数据冲突。

简言之,这是由于指令从缓存上被取得的顺序和指令中对缓存访问的顺序不一致导致的问题。

ARMv4 架构从设计上没有让单个指令同时包含读写功能,且让指令在单个时钟周期内被执行,这确保了 load、store 类型指令不会产生前后的读写冲突问题。即将流水线关注到了指令的取得和译码过程上,同时将指令执行过程原子化。这一设计避免了指令重排的问题。

此后新版本的 ARM 架构继续沿用从设计上确保读写前后的一致性的思想,并设计了宽松的内存模型(Relaxed Memory Models),同时使用了 MESI 协议设计 data cache,确保了在单线程(即缓存永远处于独占状态)情况下依然维持原指令顺序执行下对总线访问顺序的一致性^[6]。

MESI 协议也逐渐成为处理器 data cache 设计的业界标准,从古老的 Intel Pentium 处理器到较新的 ARMv8 架构均有其应用^[2,5]。

(至于具体的时钟消耗等请参见 3.3。此外,流水线涉及到了一个有趣的问题,请参见 3.5.3。)

此外,ARMv4 架构所支持的 32 位的 arm 指令集和 16 位 thumb 指令集两组指令集支持运行期间随意切换使用,并依靠状态寄存器(下文称 cspr 寄存器)的 bit5(下文称 t 位)标记当前正在执行的指令集^[3]。不过,虽然 cspr 寄存器的 t 位可以修改,但直接修改会导致死机。

如果想要进行指令集切换,则必须使用 arm 指令集和 thumb 指令集共有的 BX(Branch and eXchange)指令进行跳转。任何跳转(即有新值写入 r15 寄存器)都会导致一次流水线重置,然而只有 BX 指令会根据传入的目标地址切换指令集^[3]:

1. 传入的目标地址 bit0 为 0 切换为 arm 指令集。
2. 传入的目标地址 bit0 为 1 切换为 thumb 指令集。

不过无论是 32 位的 arm 和还是 16 位的 thumb 指令集都必须内存对齐(具体请参见 3.5.1),且指令本身定长,因此任意指令实际所在的地址的 bit0 都永远不会为 1。所以此处的目标地址的 bit0 只是作为参数而使用,实际上跳转到的地址依然是对齐的地址,bit0 的值并不会影响所跳转到的地址^[3]。

此外,在中断被触发后,处理器会在跳转至中断向量后强制切换为 arm 指令集^[3](中断的还原过程请参见 3.4.5)。

3.2 对比 ARM 架构与 X86_64 架构的栈机器

若不考虑软件的二进制程序接口的具体实现(如 C/C++ ABI),一般情况下对于处理器而言栈本质上不过是一个指向位于处理器地址空间内的某个连续地址的某个重要的一端的地址的寄存器,一般其称之为栈顶指针寄存器。

而根据栈本身的定义,弹栈、压栈则可视作由:

- a. 数据读、写。
- b. 对栈顶指针寄存器进行数据宽度的加减操作。

两部分操作构成。

在 x86_64 架构中,该栈顶指针寄存器称为 ESP(32 位模式下或 64 位模式低位)或 RSP(64 位模式下)寄存器^[2]。而在 ARMv4 架构中,该栈顶指针寄存器称为 r13(Stack Pointer)寄存器^[3]。

若考虑到二进制程序接口的实现，首先讨论 x86_64 架构（以 32 位 x86 模式举例）。

在 Intel 的文档中提到，EIP 寄存器无法直接被软件访问^[2]。它只能被控制转移指令（即 JMP, Jcc, CALL 和 RET）、中断、及异常间接控制^[2]。唯一的读取 EIP 寄存器的方法是执行 CALL 指令后从栈上读取返回地址值^[2]。而唯一写入 EIP 寄存器的方法是通过修改栈上的返回地址后执行返回指令 (RET 或 IRET)^[2]。

并且，已知 CALL func_addr 指令等价于：

```
pushl %eip
jmp func_addr
```

RET 指令等价于：

```
popl %eip
```

由此可知，虽然单纯的弹栈、压栈功能可以通过其它指令配合其它寄存器实现，且在 Intel 官方文档^[2]中 EBP/RBP 和 ESP/RSP 均被定义为 General-purpose registers（通用寄存器）。但是为了安全起见，EIP/RIP 寄存器只能通过 CALL 和 RET 指令将其值压栈/弹栈自 ESP/RSP 寄存器内的地址。因此，至少也必须将 x86_64 架构中的 ESP/RSP 寄存器视为过程调用的返回地址栈的专用寄存器。即在 x86_64 架构下，栈顶指针寄存器（ESP/RSP）的功能有其重要特殊性不能完全被通用寄存器替代。

在 ARM 架构中，控制转移相关的指令也亦涉及到了栈上行为，不过和 x86_64 架构有些许差异。

BL 指令会首先将返回地址存入 r14(LR: Link Register)寄存器（注：是存入寄存器而非存入寄存器指向的地址），此后再进行跳转^[3]。因此嵌套调用会需要额外指令将 r14 寄存器存入内存（一般是压入栈中），如下方在 main 内调用 foo1()，foo1() 内调用 foo2() 的 C++ 代码经由 ARM gcc 7.3 编译后反汇编的示例：

```
foo2():
    str    fp, [sp, #-4]!
    add    fp, sp, #0
    nop
    add    sp, fp, #0
    ldr    fp, [sp], #4
    bx     lr
foo1():
    push   {fp, lr}
    add    fp, sp, #4
    bl     foo2()
    nop
    sub    sp, fp, #4
    pop    {fp, lr}
    bx     lr
main:
    push   {fp, lr}
    add    fp, sp, #4
    bl     foo1()
    mov    r3, #0
```

```

mov    r0, r3
sub    sp, fp, #4
pop    {fp, lr}
bx     lr

```

可以看到 `fool()` 函数及 `main()` 函数通过开头和结尾的一对指令：

```

push {fp, lr}
pop {fp, lr}

```

通过栈完成了在嵌套调用下栈帧基址及返回地址的储藏和还原。（有意思的是，在 JAVA 语言的 JVM 中，动态链接称为 `dynamic link`，即每个栈帧内部所包含的该栈帧所属方法的引用；而在 ARM 处理器中，`r14` 寄存器也亦称为 `lr`，即 `link register`）

与 `x86_64` 架构的函数调用指令 `call` 会自动将返回地址压栈相比，`arm` 架构的 `bx` 则是仅仅将返回地址放入 `r14`，这导致了以栈为基础语法本质的 C/C++ 语言在一定程度上的不适应。

此外，和 `x86_64` 架构的 `EIP/RIP` 寄存器对应的 `r15` 寄存器虽然亦是指令指针寄存器，但与 `x86_64` 架构不同，它可以直接被任意指令读写^[3]（每次 `r15` 寄存器被写入时会触发指令流水线重置）。

此外，利用 `bx` 指令可直接将任意寄存器值存入 `r15` 寄存器中，同时可以根据跳转目标地址的内存对齐状况切换 32 位 `arm` 指令和 16 位 `thumb` 指令。

为配合代码块调用传参及返回的内存模型往往需要用到栈帧基址寄存器。根据标准 C/C++ ABI，栈帧基址寄存器在 C/C++ 函数调用及返回中被使用，用于指定函数执行中在栈上的可用空间基址。

在 `x86_64` 架构中，栈帧基址寄存器称为 `EBP/RBP` 寄存器，直接涉及到的只有 `enter` 和 `leave` 指令^[2]。

`enter $size` 等价于：

```

pushl %ebp
movl %esp, %ebp
subl $size, %esp

```

`leave` 等价于：

```

movl %ebp, %esp
popl %ebp

```

然而，`enter` 和 `leave` 作为 286, 386(1999 年)时代的旧指令，在现代处理器中速度反而比直接使用等价替代的指令组更慢^[6]。因此现代编译器未必会生成这两条指令。

总之可以认定，栈帧基址寄存器本身在 `x86_64` 架构中没有多少特殊性。`x86_64` 架构中栈帧基址寄存器 (`EBP/RBP`) 的功能可在实质上由任意其它通用寄存器替代。

在 `ARM` 架构中则没有定义专用的栈帧基址寄存器。仅仅在 `ARM` 架构的 C/C++ 二进制程序接口中规定栈帧基址使用 `r12` 寄存器，因此 `r12` 寄存器的别名也可称为 `fp` (`Frame Pointer`，帧指针) 寄存器^[7]。

在栈相关的操作指令方面：

在 32 位 `arm` 指令集中，`PUSH`，`POP` 指令本质不过是助记符，其实质上并没有专用的压栈弹栈指令^[3]，但是根据上文提出的栈的实质和两部分基本操作（栈本质上不过是一个指

向栈的某个重要的顶部地址的寄存器，而压栈弹栈则是由数据读写和栈顶指针寄存器自增/减)，可以得出以下内容。

a. 根据栈的增长方向，可以定义：

Ascending Stack：向地址值更大的地址增长的栈，即压栈地址值 Increment，弹栈地址值 Decrement。

Descending Stack：向地址值更小的地址增长的栈，即压栈地址值 Decrement，弹栈地址值 Increment。

b. 根据栈顶指针寄存器指向的实际内存位置，可以定义：

Full Stack：r13 指向上一次已被压栈的值的地址，即压栈先 Increment/Decrement 地址值再存入，弹栈先取出再 Decrement/Increment 地址值。

EmptyStack：r13 指向下一次将被压栈的值的地址，即压栈先存入再 Increment/Decrement 地址值，弹栈先 Decrement/Increment 地址值再取出。

以上组合配和读写操作可总结出针对不同类型的栈类型的压栈和弹栈的行为规则：

栈类型	压栈行为	弹栈行为
Full Ascending Stack	Increment Before Store	Decrement After Load
Full Descending Stack	Decrement Before Store	Increment After Load
Empty Ascending Stack	Increment After Store	Decrement Before Load
Empty Descending Stack	Decrement After Store	Increment Before Load

表 3-3 不同类型的栈类型的压栈和弹栈的行为规则

即，可以总结利用 arm 指令集的储存/读取指令功能适配各种栈的模型（该表包含了全部的块数据传输指令，可供章节 3.5.2 参考）：

栈类型	多寄存器储存作为压栈指令		多寄存器读取作为弹栈指令	
Full Ascending Stack	STMFA	STMIB	LDMFA	LDMDA
Full Descending Stack	STMGD	STMDB	LDMGD	LDMDA
Empty Ascending Stack	STMEA	STMIA	LDMEA	LDMDA
Empty Descending Stack	STMED	STMDA	LDMED	LDMDA

表 3-4 多寄存器压栈弹栈指令

单寄存器接上表：

栈类型	单寄存器储存作为压栈指令	单寄存器读取作为弹栈指令
Full Ascending Stack	str rx, [sp, #4]!	ldr rx, [sp], #-4
Full Descending Stack	str rx, [sp, #-4]!	ldr rx, [sp], #4
Empty Ascending Stack	str rx, [sp], #4	ldr rx, [sp, #-4]!
Empty Descending Stack	str rx, [sp], #-4	ldr rx, [sp, #4]!

注：

1. rx 为寄存器名。
2. Post-indexed data transfers always write back the modified base^[3]。

表 3-5 单寄存器压栈弹栈指令

例如由上表的总结可知：

```
push {r0}
str r0, [sp, #-4]!
stmfd sp!, {r0}
stmdb sp!, {r0}
```

四行命令在功能上等价，均是在 Full Descending Stack 上压入寄存器 r0。
同时可知上文 foo2() 函数亦可写为：

```
foo2():
    push    {fp}
    add     fp, sp, #0
    nop
    add     sp, fp, #0
    pop     {fp}
    bx      lr
```

对于 16 位 thumb 指令集，其包含有 PUSH/POP 指令，但其仅支持 Full Descending 栈。即：PUSH 指令功能等同 STMFD/STMDB 指令；POP 指令功能等同 LDMFD/LDMIA 指令^[3]。

3.3 ARM 架构的总线的流水线访问模拟及其时钟准确性设计

根据文档定义^[3]，ARM 架构的总线访问采用流水线设计，如果不包括协处理器，其时钟周期由三类基本时钟类型组成：

缩写	名称	说明	nMREQ	SEQ
I	Internal	没有总线访问，内部执行所消耗时钟。	1	0
N	Non-sequential	非连续地址总线读、写访问所消耗的时钟。	0	0
S	Sequential	地址连续（包含相同、增加 2 字节、增加 4 字节三种情况）总线读、写访问所消耗的时钟。	0	1

表 3-6 ARM 架构基本时钟类型

这三类时钟在 arm7tdmi 处理器硬件上由 nMREQ 和 SEQ 两根引脚决定，在改进的 arm7tdmi-s 版本中这两根引脚被更名为 TRANS[1:0]。

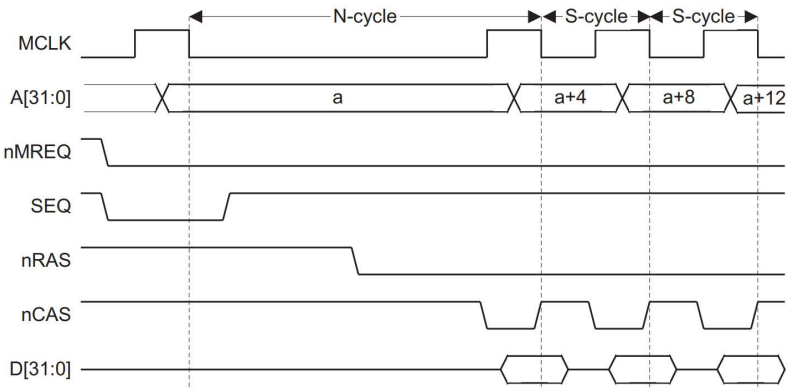


图 3-2 arm7tdmi 处理器时序图（N 时钟+S 时钟）

总线的访问由外部的 MCLK 脉冲驱动。在进入 N 时钟后，请求地址在地址总线 A[31:0] 上准备完成，提供给外部设备进行地址解码，图中的 nRAS（行访问选通脉冲）和 nCAS（列访问选通脉冲）信号为示例中一种可能的 RAM 内部信号状态，无需传入处理器^[3]。一般来说，N 时钟消耗 n+1 个时钟周期，S 时钟消耗 1 个时钟周期。

总之，在模拟器设计上可以将 S 时钟和 N 时钟的消耗交由总线控制器在处理器进行读写操作时进行计算。

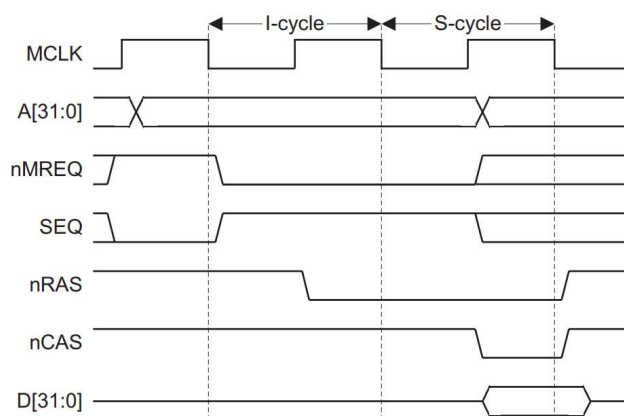


图 3-3 arm7tdmi 处理器时序图（I 时钟+S 时钟）

此外，若 I 时钟后紧接 S 时钟的总线访问，由于 I 时钟内地址总线处于空闲状态，为了辅助目标设备提前解码地址，地址总线将可能提前提供地址。这称为 Merged I-S 时钟^[3]（合并的 I 和 S 时钟）。

可以总结，数据交换产生的所有 S/N 类型时钟消耗的实际时钟数量可能由：

- 是否支持更快速的连续地址读写及非连续读写消耗的准备时钟数量，如搭配了流水线缓存。
 - 对象设备完成读写本身需要消耗的时钟。
 - 数据总线的宽度，如 16 位数据线获取 32 位数据须执行两条指令进行两次访问。
 - 对象设备被占用情况，如图形处理器在非 V-Blank 和 H-Blank 期间占用 VRAM。
- 所决定。这些都是设计处理器模拟器的总线控制器需要考虑到的可能情况。

为了将时钟消耗更好的和逻辑过程结合起进行模拟，此处总结一条 arm 指令的执行由三部分组成：

- 内部执行运算，没有产生总线访问。
- 因指令功能导致的数据读写，须通过总线访问数据。
- 获取下一条指令，须通过总线读取。

其中获取下一条指令部分在 arm 架构中可总结并定义三类：

- PR: Pipeline Reload，即流水线重置。
当 r15 被指令修改后发生，由于产生了跳转，指令流水线需要从新的执行地址补满。由于指令流水线有三段，因此：
$$PR = 1N + 2S \quad (1)$$
- SIF: Sequential Instruction Fetch，即连续指令获取。
当 r15 没有被指令修改，且下一指令的地址和上一次总线访问的地址连续时发生。

$$SIF = 1S \quad (2)$$

3. NIF: Non-sequential Instruction Fetch , 即非连续指令获取。

当 r15 没有被指令修改, 但下一指令的地址和上一次的总线访问的地址不连续 (如上一条指令的功能读/写了任意内存位置)。可得:

$$NIF = 1N \quad (3)$$

而因指令功能导致的数据读写读写部分, 在 arm 架构中也可以定义:

1. SDA: Sequential Data Access , 即连续数据访问。

$$SDA = 1S \quad (4)$$

2. NDA: Non-sequential Data Access , 即不连续数据访问。

$$NDA = 1N \quad (5)$$

例如任意位置写入一串长度为 $n(n \in \mathbb{N} \wedge n \neq 0)$ 字节的数组 ARR, 其过程消耗时钟:

$$ARR_{cyc} = NDA + \left(\left\lceil \frac{n}{4} \right\rceil - 1\right) SDA \quad (6)$$

任意一条指令 INS 消耗的时钟可以写成 $(x, y, z \in \mathbb{N})$:

$$INS_{cyc} = \begin{cases} xI + yNDA + zSDA + PR, & \text{If modified r15} \\ xI + yNDA + zSDA + SIF, & \text{Else if no bus access} \\ xI + yNDA + zSDA + NIF, & \text{Else} \end{cases} \quad (7)$$

总结之, 将 arm 架构文档中所提供的每个种类的指令所消耗的时钟^[3]通过上式分解并制成如下表格:

指令类型	I	NDA	SDA	PR/SIF/NIF
Branch and Exchange (BX)	0	0	0	PR
Branch and Branch with Link (B, BL)	0	0	0	PR
Software Interrupt (SWI)	0	0	0	PR
Normal DP	0	0	0	SIF
DP with register specified shift	1	0	0	SIF
DP with r15 written	0	0	0	PR
DP with register specified shift and r15 written	1	0	0	PR
PSR Transfer	0	0	0	SIF
Multiply (MUL)	m	0	0	SIF
Multiply-Accumulate (MLA)	m+1	0	0	SIF
Multiply Long (MULL)	(m+1)	0	0	SIF
Multiply-Accumulate Long (MLAL)	(m+1)+1	0	0	SIF
Normal SDT Load (LDR)	1	1	0	SIF
SDT Load with r15 written (LDR r15)	1	1	0	PR
SDT Store (STR)	0	1	0	NIF
Normal HFSDT Load (LDRH)	1	1	0	SIF
HFSDT Load with r15 written (LDRH r15)	1	1	0	PR
HFSDT Store (STRH)	0	1	0	NIF
Normal BDT Load (LDM)	1	1	n-1	SIF
BDT Load with r15 written (LDM r15)	1	1	n-1	PR

BDT Store (STM)	0	1	n-1	NIF
Single Data Swap (SWP)	1	1+1	0	SIF

DP: Data Processing

SDT: Single Data Transfer

HFSDT: Halfword and Signed Data Transfer

BDT: Block Data Transfer

n: 块数据传输的传输量（寄存器数量）

m: 完成乘法所需的 8 位乘法器数组消耗的时钟周期数，由乘数 Rs 的值决定，其取值如下：

m=1: 乘数 Rs 的位[32:8]全部为 0 或 1。

m=2: 乘数 Rs 的位[32:16]全部为 0 或 1。

m=3: 乘数 Rs 的位[32:24]全部为 0 或 1。

m=4: 其它情况。

表 3-7 ARM 架构基本时钟类型

上表的结果和文档中对应指令消耗的实际时钟数相等。时钟模拟通过该表可以较为优雅的合并进功能实现部分已经编写完成的模拟器内相对应行为的过程中。

3.4 一个 C++的实现方案

3.4.1 解码

Arm 架构的指令可以较为容易的使用划分 bit 位区域的方式进行分解。在 C++中则可以利用 union, struct, 和 bit field 三者进行高效率且安全的二进制 bit 位区域分解，C++ 编译器会在编译过程中生成 bit mask。示例如下：

```
#include<cinttypes>
union X{
    struct { unsigned int a:4, b:4; };
    struct { unsigned int c:5, d:2; };
    std::uint8_t val;
}x;
int main(){
    x.a = 0;
    x.c = 0;
    return 0;
}
```

其编译后的 x86 汇编代码如下，可见 x.a=0 及 x.c=0 自动被转换为对应的位运算 $x = x \& 0x11110000$ 和 $x = x \& 0x11100000$ 。

```
x:
    .zero 4
main:
    pushl %ebp
    movl %esp, %ebp
```

```

movzbl x, %eax
andl $-16, %eax
movb %al, x

movzbl x, %eax
andl $-32, %eax
movb %al, x

movl $0, %eax

popl %ebp
ret

```

使用这一方法可以避免易错且繁冗的手工编制 bit mask 过程，同时没有额外的运行效率损耗（关于解码方案的运行期效率优化请参见章节 3.7.1）。但使用需注意运行环境的大小端序情况，运行环境的端序决定了联合体 X 内的匿名结构体内的成员的二进制顺序，上例为小端序情况，左侧为高位，即成员变量 a、c 为高位。

3.4.2 流水线

Wasm_arm 会在每条指令执行结束后 fetch 流水线的下条指令^[3]，本模拟器根据指令集不同（32 位的 ARM 指令集或 16 位 Thumb 指令集）和流水线重置的触发情况分别调用：

1. PipelineFetch16()
2. PipelineFetch32()
3. PipelineReload16()
4. PipelineReload32()

四个函数。对于 PipelineReload 总是会消耗 $PR = 1N + 2S$ 个时钟、对于 PipelineFetch 则要根据指令具体情况而定。

3.4.3 寄存器

在 arm 架构下，处理器可能会进入各种模式，每个模式都有各自的部分特定寄存器组。其中 usr 模式和 sys 模式共用一个寄存器组^[3]。

每次进行寄存器访问时，如果都通过模式访问特定寄存器组，那么要么需要额外判断模式直接访问，要么需要多一层指针进行间接访问。二者效率都不高，考虑到寄存器访问是多数情况而模式切换是少见情况，wasm_arm 只会在模式切换时将\$\$\$寄存器组内容存入当前模式的寄存器组中（实际仅使用被命名为\$\$\$的运行用寄存器组），运行效率会略微提高。此外，usr 模式和 sys 模式规定没有 spsr 寄存器，如果强制访问在实际芯片上将会访问到 cpsr 寄存器^[8]，因其是少见情况（违反规定的行为），在设计上可以使用指针处理这一情况。

3.4.4 ALU

由于实际的 ARM 架构的处理器 ALU 部分包含的功能是保密未知的，在软件模拟中只能通过将指令的计算相关部分按照传统的对处理器的理解拆出并封装。

按照一般理解，一个 ALU 应包含如下功能：

1. 计算过程/结果的可能影响状态寄存器的特定比特位，如修改状态寄存器的 Negative, Carry, Zero, overflow 位。
2. 基本数字计算功能，如 Add, Subtract。
3. 移位功能，如 Logic Left/Right Shift, Arithmetic Right Shift, Rotate。
4. 位运算功能，如 And, Or。

一般来说，乘法电路因其复杂度较高及消耗时钟数较长往往是单独于 ALU 的。且在 ARM 架构中，除了专门的乘法指令（MULL, MLAL）以外的其它任何指令也没有使用到乘法部分^[3]。模拟器的 ALU 可以不包含乘法部分，但为了代码整洁性，包含亦可。

此外，虽然 MOV 指令没有严格意义上的运算过程，但是该指令的参数值会影响到状态寄存器的 Negative 和 overflow 位^[3]，因此 ALU 需要包含 MOV。

总之，伴随着对指令的理解和开发期的优化，该模拟器 ALU 最终包含以下四组功能：

1. SRNZ, SRNZ64（判断并调整状态寄存器）
2. AND, EOR, ORR, MOV, MVN, BIC
3. ADD, ADC, SUB, SBC
4. LSL, LSR, ASR, ROR

3.4.5 异常中断

一般情况下的处理器软件及硬件中断在 arm 架构下统称为异常（Exception）。为避免和 C++ 的异常、x86_64 架构下的处理器异常等概念相混淆，下文将统称中断。

Arm 处理器的中断向量表一般位于地址空间最顶部，但在使用 MMU 后可能产生变化。当中断被触发时，处理器会进入 arm 指令集模式，并跳转执行中断向量处指令。其中断向量表如下：

地址	优先级	中断	进入模式	Cpsr 标志位调整	退出指令
BASE+00h	1	rst : Reset	svc	I=1, F=1	n/a
BASE+04h	7	und : Undefined Instruction	und	I=1, F=F	movs pc, r14_und
BASE+08h	6	swi : Software Interrupt	svc	I=1, F=F	movs pc, r14_svc
BASE+0ch	5	pfa : Prefetch Abort	abt	I=1, F=F	subs pc, r14_abt, #4
BASE+10h	2	dta : Data Abort	abt	I=1, F=F	subs pc, r14_abt, #8
BASE+14h	?	[Address Exceeds 26bit]	svc	I=1, F=F	n/a
BASE+18h	4	irq : Normal Interrupt	irq	I=1, F=F	subs pc, r14_irq, #4
BASE+1ch	3	fiq : Fast Interrupt	fiq	I=1, F=1	subs pc, r14_fiq, #4

注：BASE+14h 处中断未被计入 arm 官方文档，上表中的触发特性来自于实机的验证结果。此外 rst 中断触发后 r14_svc 寄存器的值是不可信的。

表 3-8 ARM 架构中断表

处理器在接收到中断后会^[8,9]：

1. 将下条指令的地址存入 r14 寄存器。若是在 arm 指令执行中中断被触发，处理器会将 r15 寄存器的值减 4 存入 r14 寄存器；若是在 thumb 指令执行中被触发，处理器会将 r15 寄存器的值根据异常类型做出不同调整后存入 r14 寄存器。
2. 复制 cpsr 寄存器的值到中断目标模式下的 spsr。
3. cpsr 的指令集模式标志位(mode bits) 修正为 arm 指令集状态。
4. r15 寄存器跳转至对应的中断向量执行。（跳转等同于对 r15 的修改，二者都会导致流水线重载）。

对于软中断和硬中断的区别，此处以 irq 中断为例。

当 irq 在 thumb 指令执行中被触发时，r15 的值会不做修改直接存入 r14_irq。且 irq 中断的退出指令为：

```
subs pc, r14_irq, #4
```

加上 arm 中 r15-4 存入 r14_irq。根据 ARM 指令集及 Thumb 指令集的流水线：

	arm 指令集	thumb 指令集
1. [Fetch]	r15	r15
2. [Decode]	r15 - 4	r15 - 2
3. [Execute]	r15 - 8	r15 - 4

表 3-9 ARM 架构流水线

通过以上可知 irq 中断触发时无论是那个指令集，最后都实际上返回到了正在执行的指令上，fiq, pfa, dta, fiq 亦同理，此处不再详细写出。而这些中断是硬中断，换句话说是外部中断。与软中断不同的是，软中断本身不过是一条被完全执行的命令，中断退出时自然需要返回到下一条指令；而硬中断随处理器的时钟触发传入处理器，并立即终止掉了当前指令的执行，中断退出时则需要返回中断触发时正在执行的指令。

上文在具体步骤中的“根据异常类型”，指的即是软硬中断退出后跳转回的地址的不同，加上 arm 指令集指令宽度与 thumb 指令集的指令宽度不同，二者合并所导致的问题。

因为如果处理器不在此处进行区分对待，那程序员就只能针对 arm 指令集和 thumb 分别选取退出指令^[3]，例如 arm 指令执行中触发的 irq 中断的退出指令可能会是：

```
subs pc, r14_irq, #4
```

而 thumb 指令执行中触发的 IRQ 中断的退出指令则可能会是：

```
movs pc, r14_irq
```

这迫使了在中断代码中判断指令集，产生不必要的运行期消耗。

3.5 实现中遇到的问题

3.5.1 跨边界地址访问和内存对齐

在 32 位的 arm 架构中技术文档^[3]中有规定：

数据名称	数据宽度	对齐要求
Word	32 位	内存地址必须对齐 4 字节边界
Halfword	16 位	内存地址必须对齐 2 字节边界
Byte	8 位	内存地址可以放置在任意字节的边界

表 3-10 ARM 架构内存对齐要求

在本文中不符合以上任一规定的地址访问均被视为跨边界地址访问。跨边界地址访问下电路特性导致的特殊情况也是在较为精确模拟的需要下值得考虑的部分，尽管跨边界地址访问是不合手册规定的（这也导致了在 C/C++ 语言中由不同数据宽度变量组合构成的结构体实际占用的空间比数据宽度的总和大）。

Arm 架构属于冯·诺伊曼架构，数据总线同时用于传递指令数据和值数据。

对于指令数据而言，跨边界地址访问用于切换指令集，内容在章节 3.1 中涵盖，此处不再赘述。

对于值数据而言，只有 load 功能、store 功能和 swp 指令会专门访问值数据。Swp 指令行为本身是 ldr 指令后紧跟着 str 指令且其特性与这两条指令顺序执行一致，故此处不讨论 swp 指令。

此外，需要讨论的主要包含了两个情况：

1. 值数据小于数据总线宽度的情况。

即 halfword 类型和 byte 类型的读写如何使用数据总线的问题。

对于一个处理器而言，它在物理上的总线宽度是固定的，（如 32 位的冯诺依曼架构处理器可能会有 32 根数据总线连接到电路板上），实际数据宽度则往往通过引脚信号传出。本文的 arm7tdmi 处理器则可通过 MAS[1:0] 针脚判断是否是属于小于其数据线宽度的访问。

2. 值数据的跨边界地址访问。

首先，涉及到单个值数据读写的指令总结共包含：

值类型	Load 类型指令	Store 类型指令
Byte	LDRSB, LDRB	STRB
Halfword	LDRSH, LDRH	STRH
Word	LDR	STR

表 3-11 ARM 架构的 Load Store 类型指令

由于 arm 架构的寄存器为 32 位，其中 LDRSB 指令和 LDRSH 指令专门用于针对有符号 byte 和 halfword，判断其符号位并将其扩展存入 32 位的寄存器中。除了该行为之外，LDRSB 指令在功能上等同 LDRB，LDRSH 指令则等同 LDRH。

例如 LDRSB 指令的 C++ 代码的实现可以写成：

```
std::uint32_t val = bus->R8(addr, false);  
registers[instruction.hdtr.rd] = val & 0x80 ? val | 0xFFFFFFFF00 : val;
```

加上跨边界只影响到 Word 类型（未对齐 4 字节边界）和 Halfword 类型（未对齐 2 字节边界）的读写。则值数据的跨边界地址访问实际上包含以下几种情况：

1. LDR 指令读取未对齐 4 字节边界数据^[8,9]：
会产生 rotate，其 C++ 实现为：

```
auto val = bus->R32(addr);
auto shift = (addr & 0b11) * 8;
val = (val >> shift) | (val << (32 - shift));
```

2. STR 指令写入未对齐 4 字节边界数据^[8,9]：
没有不良结果。

3. LDRH 指令读取未对齐 2 字节边界数据^[8,9]：
如果 bit0 为 1，会读入一个错误的值，如 C++ 实现为：

```
auto val = bus->R16(addr, false);
if (addr & 1) val = (val >> 8) | (val << 24);
```

4. STRH 指令写入未对齐 2 字节边界数据^[8,9]：
会产生无法预测的行为。

3.5.2 块数据传输指令的子阶段

由于 arm 架构总线访问的流水线特性，一般 S 时钟地址访问消耗 1 时钟完成，然而，块数据传输指令由于访问了多处地址，不仅会占用多个时钟周期，并且会在访问期间会阻塞数据总线，其具体的执行过程有一些值得注意的地方。（包含的指令请参考章节 3.2 中表格，此外章节 3.2 提到了一种块数据传输指令的用法）

此处以 STMDB 指令为例。从表面上看，STMDB (Store Multiple Decrement Before) 指令的功能是地址先减小再存入，即：地址减小、存入地址、地址再减小、再存入下一个地址、如此往复至块数据写入完成。然而实机的具体的访问过程违背了该直觉。实机将提前计算好最后一个被存入的地址，反过来以地址增加的方式进行存入^[10]。即可以理解为，在 arm 架构处理器中块数据传输指令对总线的访问没有真正意义上的 decrement。STMDB, STMDA, LDMDb, LDMDA 四枚指令本身对总线访问的顺序实际上等同于提前计算好了最终地址并逆向使用寄存器组的 STMIA, STMIB, LDMIA, LDMIB。

即，从总线访问情况中观测来看，指令：

```
stmdb r0, {r1-r3}
```

会在寄存器值为：

r0	r1	r2	r3
0x0000000C	0x00000008	0x00000004	0x00000000

表 3-12 寄存器示例值

的情况下以以下的顺序访问总线：

```
WRITE 00000000: 0x00000000
↓
WRITE 00000004: 0x00000004
↓
WRITE 00000008: 0x00000008
```


由于目标块寄存器组的组成选取采用二进制掩码直接作为指令参数，模拟器可以提前算出寄存器组参数的汉明重量并与地址相减以实现该特性。

该特性被任天堂公司的 GBA 游戏利用作为一个模拟环境的检测手段^[9]。GBA 游戏机中包含四组 DMA，其中每组 DMA 包含四个参数寄存器，每个寄存器为 16 位。每组的最后一个寄存器为该 DMA 的控制寄存器，对 DMA 控制寄存器的 Enable 位(bit15)写入 1 会立即触发数据传输行为。配合块数据传输子周期内实机处理器 STMDB 指令对地址的特殊访问顺序，成为了任天堂在 GBA 游戏中检测模拟器环境的手段。

在游戏 Classic NES Series 中，错误的块数据传输顺序会导致在参数尚未传入前三个参数寄存器时就写入了第四个控制寄存器，DMA 将提前被触发，导致游戏进入错误界面，阻止玩家继续进行游戏。



图 3-4 检测到处于模拟器环境下的错误画面

3.5.3 正确模拟流水线的重要性

Arm 架构中在设计中避免了读→修改→写的指令。该设计思想避免了类似 MIPS 架构的 R3000 处理器中常见的指令重排，使得编译到 arm 架构的编译器无需考虑写后读，读后写，写后写的数据相关冲突问题。部分早期的模拟器仅将 r15 寄存器的值指向正确的位置（即最后进入流水线的指令的地址）。任天堂利用了这一思维方向，设计了一处针对没有正确模拟流水线的模拟器的陷阱^[11]。

此处以任天堂发行在 GBA 游戏机上的 Classic NES Series 游戏举例，其中有一段代码为：

```
06000260h: mov r1, #0
06000264h: add lr, pc, #8
06000268h: ldr r0, [$06000260]
0600026Ch: str r0, [lr, #0]
06000270h: mov r1, #255
06000274h: mov r1, #255
```

代码段执行完的 r1 寄存器值被用于判断程序运行环境的合法性。0600026Ch 处指令将地址 06000260h 的指令拷贝入地址 06000274h（即 mov r1, #0）。

然而，在指令执行到 0600026Ch 时，06000274h 已经进入了指令流水线的 fetch 阶段，对其内存中的内容修改并不会改变已经进入流水线等待执行的指令。

因此在真正机器上，本段代码执行完毕后 r1 寄存器最后的值为 0；在没能正确模拟流水线的模拟器中，r1 寄存器最后的值为 255。

3.6 一个半自动化的 ARM 指令集架构模拟器的测试方法

3.6.1 ARM 处理器的有限状态机建模

Arm 架构处理器的状态组成包含有限个寄存器，每个寄存器存储的值为有限的大小，可访问的全部地址也亦是有限的 2^{32} 个字节。因此 arm 架构处理器配合其周边系统组成的独立机器可被视为一个有限状态机 FSM (finite state machine)，根据 FSM 一般式：

$$M = (S, \Sigma, \delta, S_0, F) \quad (1)$$

其中，有穷状态集 S 的任意状态 $s \in S$ 包含了：

1. pc : Program Counter，即 r15 寄存器，是一个 32 位二进制数，用于确定程序执行到的地址位置，其中 $pc \in \{0, \dots, 2^{32} - 1\} \cup \{\text{halt}\}$ ，halt 则用于表示停机状态。
2. $SPSR$: 由五个子集构成，包含：

$$SPSR = (SPSR_{FIQ}, SPSR_{SVC}, SPSR_{ABT}, SPSR_{IRQ}, SPSR_{UND}) \quad (2)$$

其中每个子集都与 cpsr 寄存器的构成相同。

3. $CPSR$: 即 cpsr 寄存器，由处理器的运行状态标志位构成，其中 m 为 5bit，其余为 1bit，即 $m \in \{0, \dots, 2^5 - 1\}$ ， $n, z, c, v, i, f, t \in \{0, 1\}$ 。

$$CPSR = (n, z, c, v, i, f, t, m) \quad (3)$$

4. REG : 由全部七个模式下的一般寄存器组成的集合（其中 USER 模式和 SYS 模式共享同一组寄存器）：

$$REG = (USERSYS_{REG}, FIQ_{REG}, SVC_{REG}, ABT_{REG}, IRQ_{REG}, UND_{REG}) \quad (4)$$

其中 $USERSYS_{REG} = (r_0, \dots, r_{14})$ ， \dots ，且对于任意寄存器 $r_n \in \{0, \dots, 2^{32} - 1\} (n \leq 15, n \in \mathbb{N})$ 。

5. MEM : 内存状态的集合，受 arm 架构可访问的地址空间限制，其包含 2^{32} 个字节。

即：

$$s = (pc, SPSR, CPSR, REG, MEM), \forall s \in S \quad (5)$$

其状态转换函数 $\delta: S \times \Sigma \rightarrow S$ 通过在状态 s 时执行合法指令集合 Σ 内的某一指令 $x \in \Sigma$ ，即 $\delta(s, x)$ ，使得状态 s 转换为状态 s' 。若 wasm_arm 平台的 arm 模拟器所模拟的处理器行为为 δ_e ，实际的物理处理器的行为为 δ_r ，那么，当：

$$\forall s \in S: \delta_r(s, x) = \delta_e(s, x) \quad (6)$$

时，可以认为该模拟器完全正确的对指令 x 进行了模拟。反之，若：

$$\exists s \in S: \delta_r(s, x) \neq \delta_e(s, x) \quad (7)$$

时，可以认为该模拟器在 s 状态下的指令 x 的模拟逻辑中包含错误。

3.6.2 半自动化半随机测试集生成

由上文可知，进行测试需要初始化的参数包含两大部分，状态 s 和指令 x 。设待测指令 x 测试前准备好的待测状态为 s' ，待测指令执行完毕后状态切换为 s'' ，那么可得出：

$$\delta_e(s', x) = (s'') \quad (8)$$

换言之即：

$$\delta_e(pc', SPSR', CPSR', REG', MEM', x) = (pc'', SPSR'', CPSR'', REG'', MEM'') \quad (9)$$

然而，如果对所有可能的状态 $\forall s' \in S$ 和所有的指令 $\forall x \in \Sigma$ 全部情况进行生成并测试，由于集合过于庞大，从时间和空间角度考虑是不现实的，因此需要对这一过程进行优化。

对于状态而言，由于不同类型的指令功能仅涉及到部分确定相关的状态，而剩余的状态则与该指令无关不应受其影响。因此只有部分状态会与对应指令的状态转移函数互相影响需要进行随机生成并和实机交叉检验。剩余的部分状态由于不应当被改变，因此仅需将其和执行前状态进行对比。根据指令功能类型，可制定实际测试的状态空间，例如当指令 y 为 Branch and Exchange 类型时，仅 pc 会受到影响，即：

$$\delta_e(pc', \dots, y) = (pc'', \dots) \quad (10)$$

对于指令而言，armv4t 架构包含有 32 位定长的 arm 指令集和 16 位定长的 thumb 指令集，二者均未占满全部 2^{32} 和 2^{16} 的取值空间，指令测试须根据指令解码表针对其中的合法指令进行。此外，考虑到寄存器编号类型的参数和立即数不影响 wasm_arm 模拟器代码执行的逻辑流，因此对于这两者仅取一个合法的定值进行测试。此外，本测试不针对协处理器相关指令。

该测试的指令范围由指令解码表可得出：

Instruction Type	Test Range(2^n)
Data Processing / PSR Transfer	18
Multiply	2
Multiply Long	3
Single Data Swap	1
Branch and Exchange	0
Half Data Transfer Register Offset	6
Half Data Transfer Imme Offset	14
Single Data Transfer	17
Undefined	0
Block Data Transfer	21
Branch	1
Software Interrupt	0

表 3-13 测试的指令范围

共计须检测 2506835 条指令，假设实机处理器运行频率为 16.78MHz，若不考虑前后的验证及状态准备相关代码逻辑，且假设平均每条指令消耗 3 周期运行，则指令执行共须花费约 0.4274 秒。若假设前后的验证及准备相关代码逻辑消耗 300 周期，则共需花费约 43.1694 秒，消耗预估处于可接受范围内。

从另一个角度来看，代码路径构成了一个有向无环图 DAG (directed acyclic graph)，而其中分支选取只涉及指令和状态，测试优化的目的是利用尽可能少的访问次数的同时，尽可能完全的走完图中全部节点。如下图是模拟器 DAG 的局部情景：

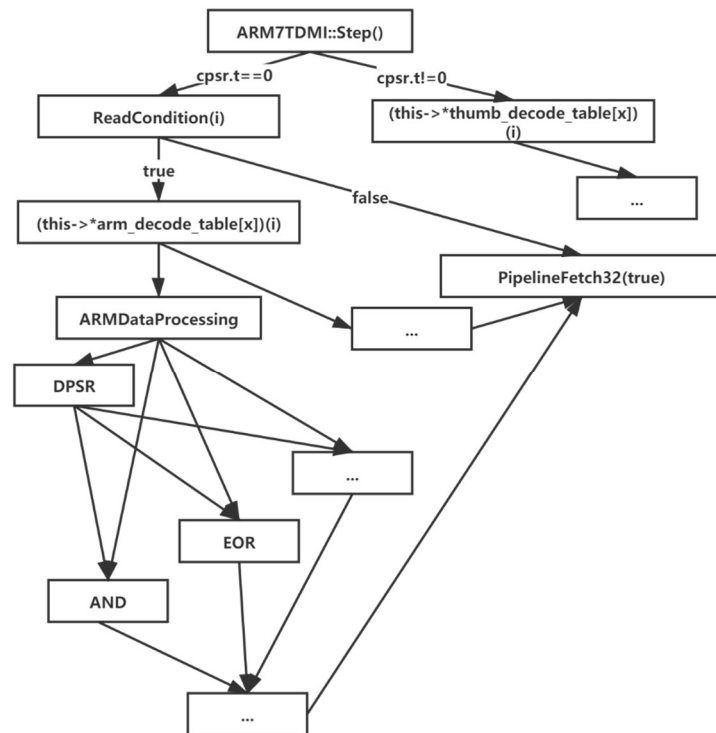


图 3-5 模拟器 DAG

3.6.3 自动化实机交叉检验

由于 wasm_arm 平台在开发过程中避免了运行平台依赖，因此可利用该 arm 模拟器模拟所模拟的处理器作为该 arm 模拟器的编译目标。本文将简述一个在 GBA 游戏机上运行该模拟器的同时利用 GBA 游戏机自身处理器对该模拟器进行交叉检验的方案。

根据章节 3.6.1 的结论可知，进行指令执行准确性验证的核心要素是完全同步的模拟处理器和实体处理器状态，而状态本身则包含了地址空间和寄存器两者。因此，第一步则应是将模拟器的状态和实机完全同步，其大致理念如下图所示：

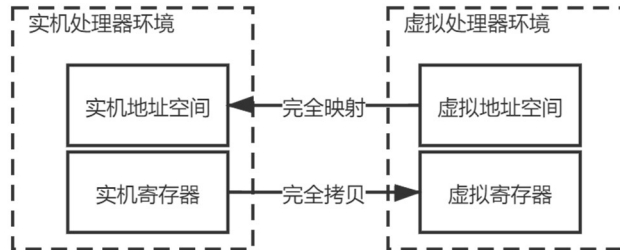


图 3-6 交叉检验的理想状态

其中地址空间的映射的读取部分可通过模拟器的总线控制器直接访问对应内存地址达成，例如其中读内存：

```
std::uint32_t R32 (std::uint32_t addr, bool isSeq){
    return *reinterpret_cast<std::uint32_t*>(
        &(reinterpret_cast<std::uint8_t*>(0))[addr]);
}
```

对于地址写入部分，为了能将实机写入内容和模拟器写入内容二者做出对比，可设计待测指令若涉及写入，则仅写入一段合法空间（例如一段任意长度的数组，下文称 Writable Area），并拦截模拟器部分的写入行为且将行为存入 Write Action Container。在模拟完全执行完毕后，将 Write Action Container 中的数据和 Writable Area 进行对比。

例如其中拦截模拟器写入行为的代码：

```
void w32 (std::uint32_t addr, std::uint32_t val, bool isSeq){
    wa->push_back(std::make_tuple(addr + 0, static_cast<std::uint8_t>(val >> 0)));
    wa->push_back(std::make_tuple(addr + 1, static_cast<std::uint8_t>(val >> 8)));
    wa->push_back(std::make_tuple(addr + 2, static_cast<std::uint8_t>(val >> 16)));
    wa->push_back(std::make_tuple(addr + 3, static_cast<std::uint8_t>(val >> 24)));
}
```

其设计图如下：

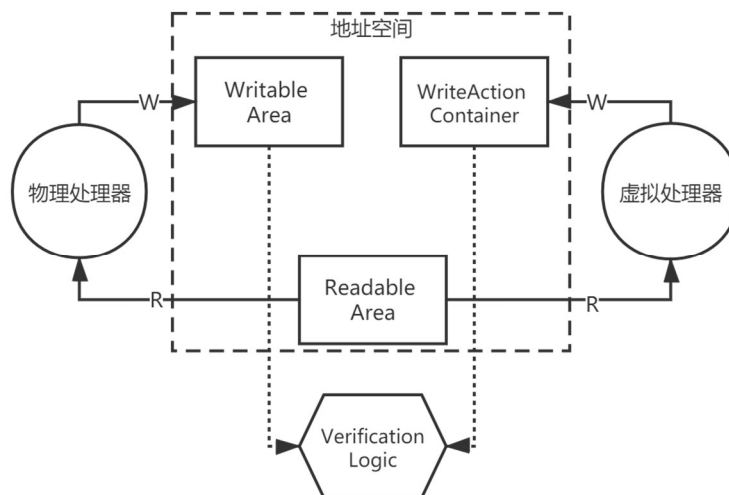


图 3-7 交叉检验的实际地址访问设计

寄存器的完全拷贝可通过内联汇编完成，例如：

```
std::uint32_t* ptr_simgreg = &(sim.registers.current[0]);
std::uint32_t* ptr_simcpsr = &(sim.registers.cpsr.value);
__asm("ldr r0, %[val]" : : [val] "m" (ptr_simcpsr));
__asm("mrs r1, cpsr");
__asm("str r1, [r0]");
__asm("ldr r0, %[val]" : : [val] "m" (ptr_simgreg));
__asm("stmia r0, {r0-r15}");
```

由于该模拟器在同一台实机上进行执行，在寄存器拷贝过程中，由于临时寄存器是必须需要的，实机的部分寄存器有可能在同步时被修改（即此处的 r0 和 r1）导致状态坍塌（即薛定谔的寄存器，对寄存器的观测行为本身需要利用寄存器，导致寄存器的状态坍塌），因此可限定寄存器 r0, r1 作为临时的 temp 寄存器，不介入模拟的验证过程，同时也不会作为待测指令的寄存器参数。

二者状态同步后便可执行二者的待测指令，由于模拟器在同一台实机上进行，因此需要先执行实机的待测指令，并将实机执行后的寄存器内容拷贝至内存中测试无关的区域进行备份，之后再执行模拟器代码。

完整的设计图如下所示：

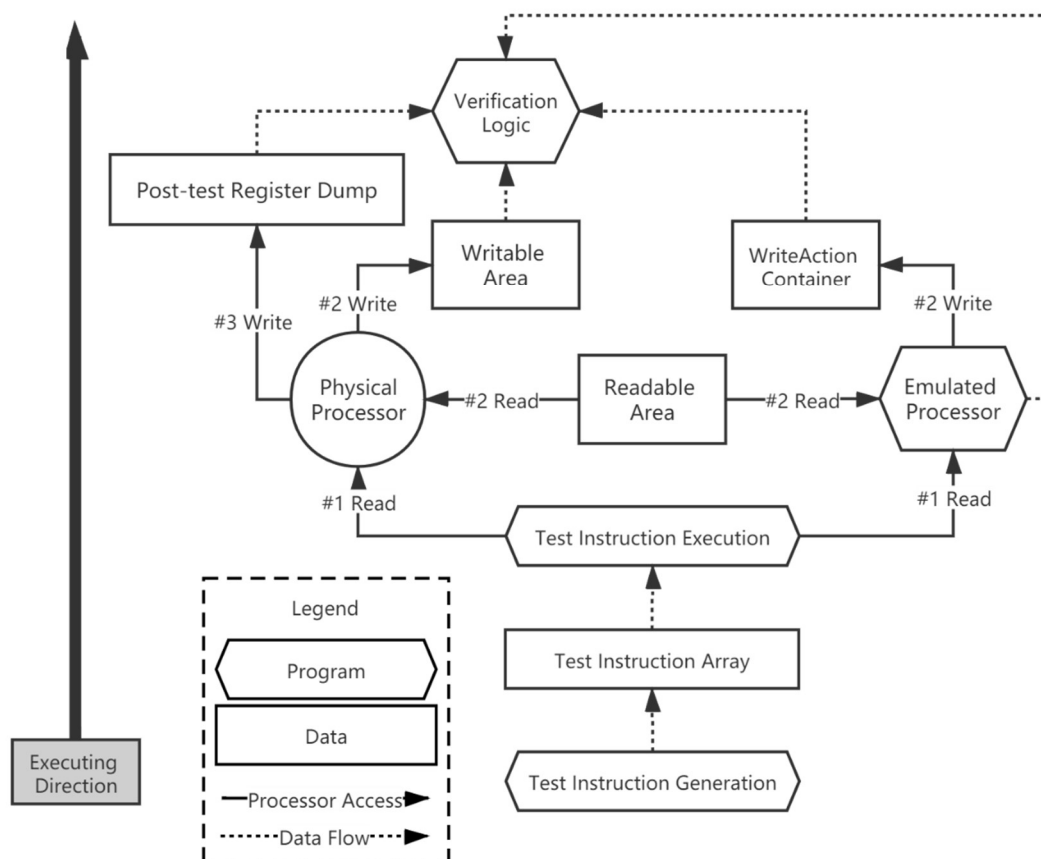


图 3-8 交叉检验的实际处理流程

例如下文测试了指令 `mov r2, #0xEE` :

```
//Fill instruction pipeline
__asm("nop"); __asm("nop"); __asm("nop");

/** TEST CASE START **/
__asm("mov r2, #0xEE");
/** TEST CASE END **/

//Obtain a copy of physical (It causes r15 to increase)
__asm("ldr r0, %[val]" : : [val] "m" (ptr_phytmp));
__asm("mrs r1, cpsr");
__asm("str r1, [r0, #64]");
__asm("stmia r0, {r0-r15}");
ptr_phytmp[15] -= 4*4;

//Temp registers are useless (r0, r1)
ptr_phytmp[0] = 0; ptr_phytmp[1] = 0;
ptr_simgreg[0] = 0; ptr_simgreg[1] = 0;
```

```
//Fill instruction pipeline
sim.Step(); sim.Step(); sim.Step();

/** TEST START **/
sim.Step();
/** TEST END  **/
```

由于指令流水线的存在，需要准备 NOP 指令填补模拟处理器。执行完毕后便可以对比二者的结果，如一致，则代表正确模拟，如下图为上文测试的寄存器对比的结果输出：

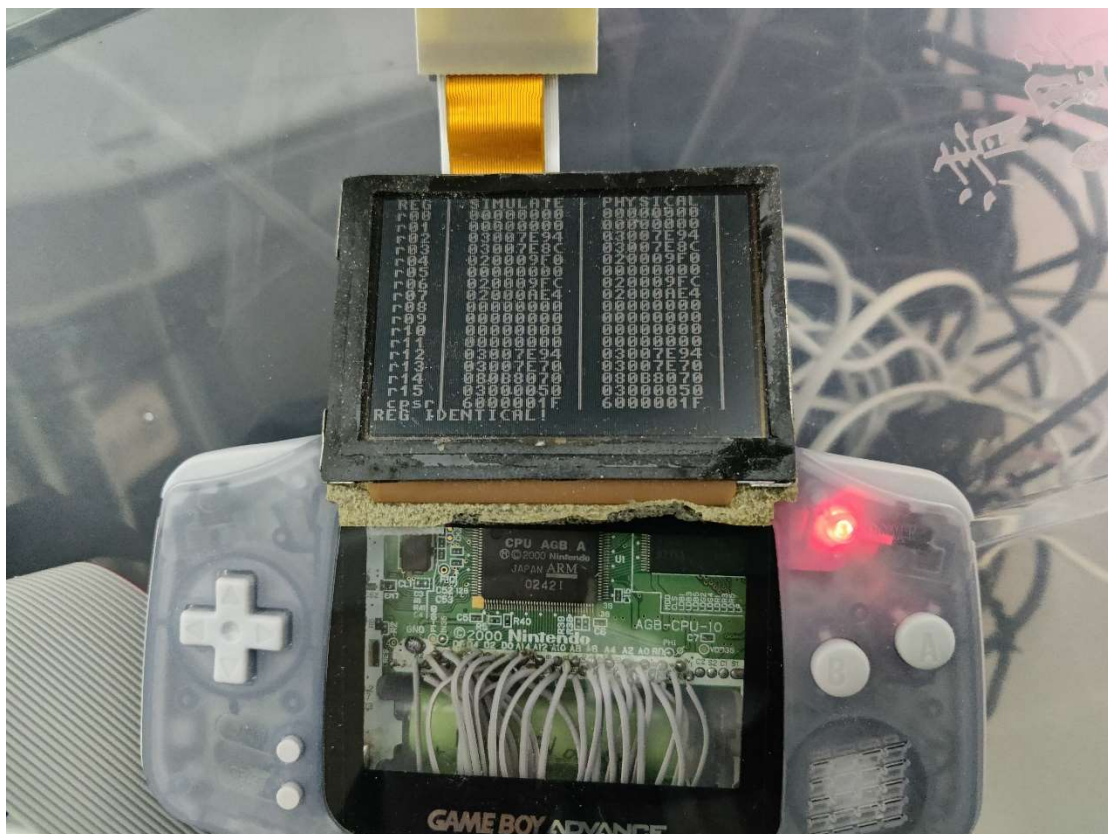


图 3-9 交叉检验的结果输出

3.7 ARM 模拟器效率优化

3.7.1 指令编译期预解码和状态码编译期预生成

在 arm 指令集中有两组指令类型组在解码表中有表面上的潜在冲突，分别是 Single Data Transfer（下文称 SDT）和 Undefined（Undefined 仅代表 2^{32} 大小的指令取值空间中符合该定义的未定义指令的其中一部分的子集，该子集可被用于定义软件实现的自定义指令，除该子集外仍有许多未定义的指令格式未被包含在内），二者解码格式如下^[3]：

BIT	31 - 28	27	26	25	24	23	22	21	20	19 - 16	15 - 12	11 - 5	4	3 - 0
SDT	Cond	0	1	I	P	U	B	W	L	Rn	Rd	Offset		
Undefined	Cond	0	1	1									1	

表 3-14 SDT 和 UND 类型指令解码对比

其中 SDT 在 bit25 的 I 参数为 1 的同时 Offset 参数内 bit4 的值亦为 1 时可能被视为 Undefined 指令。实际上，在该 I 参数传入 1 后，bit11 至 bit0 的 Offset 参数不再是一个单独的立即数，而是带有偏移量参数 Shift 的寄存器编号 Rm，且其须被解码为：

11 - 4	3 - 0
Shift	Rm

表 3-15 Offset 的指令解码

且 bit11 至 bit4 的偏移量参数 Shift 可根据 bit4 确定偏移量的实际取值方案：

BIT	11 - 8	7	6 - 5	4
Immediate Value	Shift Amount		Shift Type	0
Register Specified	Rs		Shift Type	1

表 3-16 Shift 的指令解码

然而，SDT 指令组并不支持利用寄存器作为偏移量的参数，即 bit4 只能为 0。换言之，SDT 指令组同 Undefined 指令组并不存在解码冲突。

总而言之，arm 指令集的解码过程并非简单的二进制编码模式匹配，而是在解码中包含了状态机的状态迁移过程。

该状态迁移过程可部分利用 C++模板编程在编译器进行提前解码，从而达到指令的编译期预解码。为了能减少运行时的性能损耗，在运行空间允许的情况下，应该尽可能的在编译期就能通过指令的部分特征标志位生成全部指令格式的解码方案，并在运行时直接通过传入指令的特征标志位判断并使用对应方案的数据处理流程。

那么，首先需要找到区分全部的指令格式且所需数量最少的标志位。

可以先针对每种指令类型找出指令参数的 bitmask，并在最后采用 bitmask 的 bitmask 作为标志位。其中指令参数被划分为两类：

1. 值参数部分。如 Rn, Rd, Offset 等。既无法判断出指令格式，也基本上不影响代码流程，不作为标志位。
2. 控制参数部分。如 Opcode, I, U, A, S 等。虽然无法用于判断格式，但会直接影响代码逻辑流程，为了使 C++模板生成对应逻辑流，可作为标志位。

对于 arm 指令集，根据其指令结构可知 bit31 至 bit28 为固定的 condition 位，且绝大多数的值参数位于 bit19 至 bit16、bit15 至 bit12、bit11 至 bit8、bit3 至 bit0。

总之根据指令解码表可以得出，指令 bitmask 的 bitmask 为 0x0FF000F0，有效位为 12 位，即大小为 0x1000。

在 wasm_arm 中做出了如下设计：

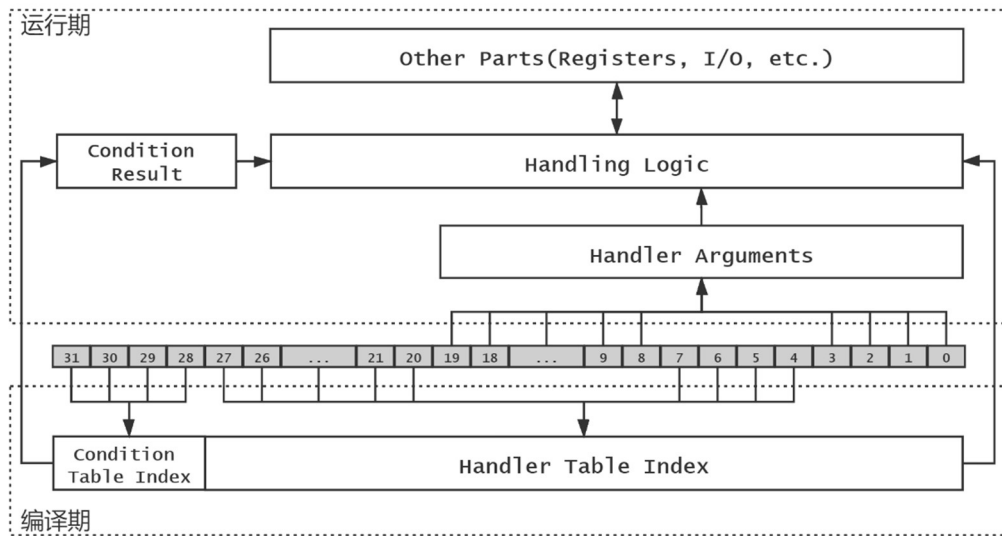


图 3-10 解码逻辑示意

使用 c++元编程，利用该“bitmask 的 bitmask”，在编译期传入标志位利用模板参数生成对应的逻辑代码流，并将函数指针放入对应下标函数数组内。

```
constexpr auto ARM7TDMI::ARMDecodeTableGeneration() ->
std::array<ARM7TDMI::ARMHandler, 0x1000> {
    std::array<ARM7TDMI::ARMHandler, 0x1000> table{};
    static_for<std::size_t, 0, 0x1000>([&](auto i) {
        table[i] =
            ARMDecoder<((i & 0xFF0) << 16) |
            ((i & 0x00F) << 4)>();
    });
    return table;
}
std::array<ARM7TDMI::ARMHandler, 0x1000>
ARM7TDMI::arm_decode_table = ARM7TDMI::ARMDecodeTableGeneration();
```

此处的 static for 利用了 C++17 版本标准库的 make_integer_sequence 生成编译期自然数序列：

```
#include <utility>
template <typename T, T Begin, class Func, T ...Is>
static constexpr void static_for_impl
    (Func&& f, std::integer_sequence<T, Is...>) {
    (f(std::integral_constant<T, Begin + Is>{ }), ...);
}
template <typename T, T Begin, T End, class Func>
```

```
static constexpr void static_for(Func&& f) {
    static_for_impl<T, Begin>(
        std::forward<Func>(f),
        std::make_integer_sequence<T, End - Begin>{ }
    );
}
```

对于 arm 指令集指令开头固定的 Cond 位，其与状态寄存器的 nzcv 位直接相关，用于判断指令是否会被执行。

```
enum CONDITION {
    EQ = 0b0000, NE = 0b0001, CS = 0b0010, CC = 0b0011,
    MI = 0b0100, PL = 0b0101, VS = 0b0110, VC = 0b0111,
    HI = 0b1000, LS = 0b1001, GE = 0b1010, LT = 0b1011,
    GT = 0b1100, LE = 0b1101, AL = 0b1110, NV = 0b1111
};
```

针对其可以直接生成 $2^4 \times 2^4$ 大小的查找表，例如：

```
constexpr auto ARM7TDMI::ConditionTableGeneration() ->
std::array<std::array<bool, 16>, 16> {
    std::array<std::array<bool, 16>, 16> table{};
    for(int i=0;i<16;i++) {
        bool n = (i & 0b1000), z = (i & 0b0100),
            c = (i & 0b0010), v = (i & 0b0001);
        table[i] = std::array<bool, 16> {
            z, !z, c, !c,
            n, !n, v, !v,
            c&&!z, !c||z, n==v, n!=v,
            !z&&(n==v), z||(n!=v), true, false
        };
    }
    return table;
}
```

对于 thumb 指令集的方案，其类似 arm 指令集，此处不再赘述。总之，根据 thumb 指令结构，其没有 Condition 位，且大多数的值参数位于 bit7 至 bit0。

3.7.2 对比 reinterpret_cast 和 Bit Shift

模拟器的内存空间以 uint8_t 数组的形式存储。而类似处理器部分等都是 32 位或 16 位的小端序设备，因此在设备与内存间进行数据交换时必然要调整字节顺序。对于如下小端序设备内的 32 位数据写入内存地址 0x00 的情形：

```
std::array<uint8_t, 0x100> memory;
uint32_t addr = 0x00;
uint32_t data = 0x01020304;
```

一般可以使用 bit shift 进行端序转换，如：

```
memory[addr+0] = (uint8_t)(data >> 0);
```

```
memory[addr+1] = (uint8_t)(data >> 8);
memory[addr+2] = (uint8_t)(data >> 16);
memory[addr+3] = (uint8_t)(data >> 24);
```

此外亦可利用 `reinterpret_cast`，作为编译期的类型 `cast`，其类似于 `const_cast`，但 `const_cast` 用于在编译期将带 `const` 修饰符的常量解释为变量从而支持常量修改，而 `reinterpret_cast` 用于在编译期将变量类型解释为新类型从而切换类型处理的法则^[12]。新类型则使用模板参数传入。如：

```
*reinterpret_cast<uint32_t*>(
    &(reinterpret_cast<uint8_t*>(&memory))[addr]
) = data;
```

将以上两种方法使用 `x86_64 gcc 9.2` 编译器编译 32 位环境且优化参数为 `O3` 时生成的汇编：

```
main:
    movl    data, %edx
    movl    addr, %eax
; 使用 bit shift 方式:
    movl    %edx, %ecx
    movb    %dh, memory+1(%eax)
    shr1    $16, %ecx
    movb    %cl, memory+2(%eax)
    movl    %edx, %ecx
    shr1    $24, %ecx
    movb    %cl, memory+3(%eax)
; 使用 reinterpret_cast 方式:
    movl    %edx, memory(%eax)
    xorl    %eax, %eax
    ret
data:
    .long   16909060
addr:
    .zero   4
memory:
    .zero   256
```

可以看出使用 `reinterpret_cast` 后效率有很大提升，使用其更佳。不过使用时须注意运行平台的端序情况，且须注意数组越界。

4 浏览器内的简单 ARM 微架构 WASM_ARM

4.1 基本设计

4.1.1 内存模型，Bootloader，以及启动初始化流程

进行单纯的处理器模拟并不能直接实现针对该处理器编译的代码的运行，实现这一目标最起码需要一个最简单的类嵌入式的简易环境，wasm_arm 平台便是该简易平台的一种实现。其包含有 32Mbyte 的连续有效地址空间，并且全部可读可写可执行（RWX）。

地址区域	用途
[0x00000000, 0x00000020)	中断向量
[0x00000020, 0x01FFFF00)	可自由使用内存
[0x01FFFF00, 0x02000000)	简易输出设备

表 4-1 WASM_ARM 大致内存区域

针对该空间可设计链接器脚本：

段	子段	说明
.reset		Bootloader 代码段所在部分，包含了启动环境初始化逻辑、中断向量表及中断的初始处理逻辑
.text		C++代码段，main 函数亦在此内
.ARM.exidx		Exception-handling table index，栈追踪和栈 unwind 时使用（如异常和 debug 的调用栈跟踪）的、包含一个经二叉树排序的各函数入口地址及其对应的 extab（exception-handling table, EHT）表地址（该表一般位于 .text 段内）的下标 [13]
.data		数据段，由于模拟环境全部地址空间可读可写，因此和真机不同初始化时无需对其进行拷贝
.bss		静态数据段，在 bootloader 中初始化为 0
.stack	__irq_stack_top__	4Kbyte 的 irq 模式栈，由于模拟环境中内存直接可视，因此在初始化时使用特殊值染色方便观测
	__fiq_stack_top__	fiq 模式，其余同上
	__svc_stack_top__	svc 模式栈，其余同上
	__abt_stack_top__	abt 模式栈，其余同上
	__und_stack_top__	und 模式栈，其余同上
	__sys_stack_top__	1Mbyte 的 sys 模式栈，其余同上
.heap		2Mbyte 的堆内存空间，对其的管理请参见章节 4.3.1
...		空
.scrout		简易输出设备

表 4-2 WASM_ARM 详细内存区域

Wasm_arm 平台的启动初始化流程:

ARM 架构处理器启动时并没有必须进行的初始化步骤,但一般来说,至少要初始化各个模式下的栈顶指针。初始化大多数针对的是 C++ 部分,最基本的是需要将 bss 段内存初始化为 0 并执行静态构造函数。

1. 处理器启动时默认执行 reset 中断,并执行 0x00000000 处中断向量处指令跳转进入 bootloader 代码内的_rst 段。
2. 遍历 bss 段内存空间,将值初始化为 0。
3. 遍历栈空间设定为特定染色值。
4. 进入各个模式下,将其 r13 寄存器初始化为对应模式的基础栈顶位置(如 irq 的对应上表的__irq_stack_top__编译期变量)。
5. 执行静态 C++ 构造函数。
6. 调用执行 C++ main() 函数。
7. main() 函数执行完毕后,触发参数为 0xffffffff 的软件中断(swi)。

Wasm_arm 平台的中断异常处理:

Bootloader 位于地址空间的顶部,并在其中同时包含中断向量表及其基本处理逻辑。

对于 und 和 swi 中断,由于本平台只是一个简易类嵌入式的演示环境,并没有特殊的设备及情况需处理,因此其只会单纯的切换回 sys 模式,并调用对应的函数指针;该函数指针可由用户自定义(如直接传入 lambda 表达式)。

对于 irq 中断,其依据 ARM-AAPCS 额外包含了一个将寄存器状态保留到栈中并再用户代码执行完后还原的过程。

4.1.2 连接 WebAssembly

本平台与环境的关系结构组成如图所示：

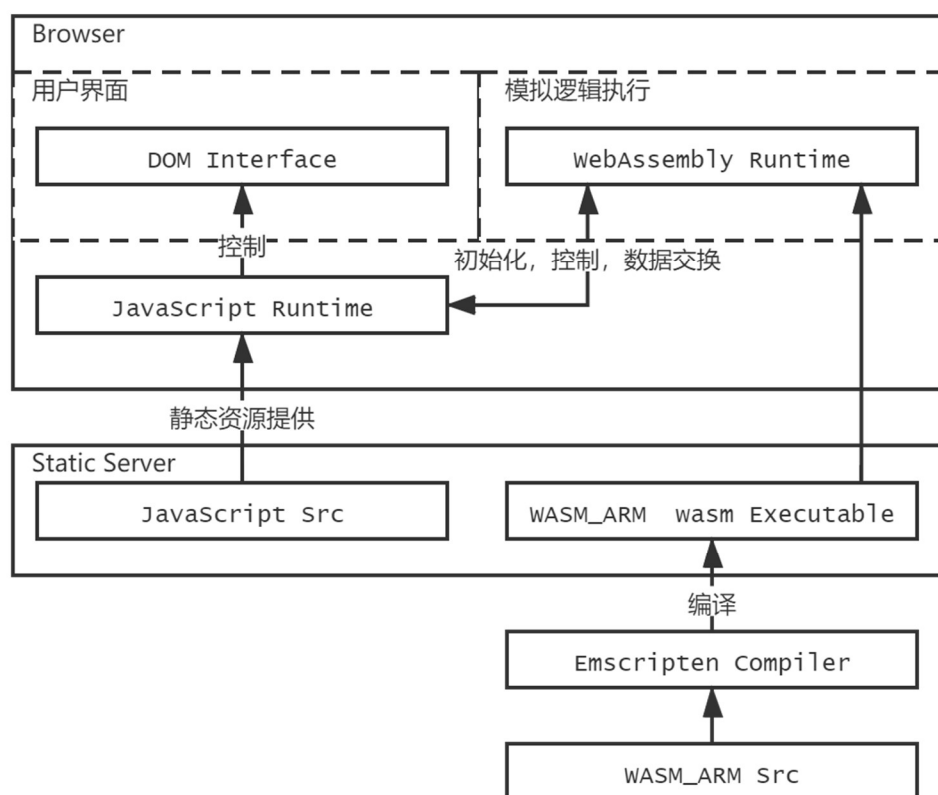


图 4-1 模拟平台开发及运行中与环境的关系

鉴于 JavaScript 操作 DOM 及静态资源的提供十分简单，本部分将省略这些内容，仅说明 JavaScript 运行时同 WebAssembly 运行时的数据交换。

由于 WebAssembly 环境（下文称 wasm 环境）可以被 Javascript 侧控制^[1]，因此可以在 Javascript 中开辟 wasm 环境的堆上空间，并利用此配合函数指针绑定用于数据传递。例如下文的 Javascript 代码利用 wasm 环境的堆内存传递了一段 Uint8 数组，并将其指针传入 wasm 环境中的函数 `passData` 并调用。此外需要注意 wasm 环境异于 Javascript 环境，传递后的内容使用完毕后需要对其进行回收：

```
function sendArr(arr){
    const arrPtr = window.Module._malloc(arr.length);
    window.Module.HEAPU8.set(arr, arrPtr);
    window.Module.passData(arrPtr, arr.length);
    const returnArr = new Uint8Array(arr.length);
    returnArr.set(
        window.Module.HEAPU8.subarray(arrPtr, arrPtr + arr.length)
    );
    window.Module._free(arrPtr);
}
```

此外，函数暴露到 Javascript 须在编译期绑定函数名与函数指针^[1]：

```
void PassData(const int& addr, const std::size_t& len);
EMSCRIPTEN_BINDINGS(my_module) {
    emscripten::function("passData", &PassData);
}
void PassData(const int& addr, const std::size_t& len){
    //
}
```

4.2 验证 ARM 的 C/C++二进制程序接口

4.2.1 函数调用过程及变量生命周期

在函数调用时，传统情况下（例如 32 位 x86 架构）C++ 会优先利用栈传递参数。但在 arm 架构和 x64 架构中，寄存器会优先用于参数传递。其中 arm 架构中，r0, r1, r2 寄存器直接用作参数传递，参数若超过三个，此后的参数则会利用 r3 寄存器回到传统的压栈传参的方法^[7]。而 x64 架构由于寄存器数量更多，前六个参数会采用 edi、esi、edx、ecx、r8d、r9d 的寄存器顺序传递，余下的参数压栈^[2]。

例如在 C++ 代码中调用 foo() 函数，传入了六个参数：

```
foo(0, 1, 2, 3, 4, 5);
```

其使用 arm gcc 8.2 编译器及 -O0 参数免去优化的编译后的结果如下：

```
mov    r3, #5
str     r3, [sp, #4]
mov     r3, #4
str     r3, [sp]
mov     r3, #3
mov     r2, #2
mov     r1, #1
mov     r0, #0
bl      foo(int, int, int, int, int, int)
```

此后，函数利用 r0 寄存器传回返回值，与 x86 的下利用 eax 寄存器十分相似。而函数内的成员变量则一般是在栈上，例如函数内的普通数组：

```
int i[4];
i[0] = 12;
i[1] = 34;
i[2] = 56;
i[3] = 78;
return i[0];
```

均可以表示为针对 fp 寄存器、即 r11 寄存器的偏移量，最后在返回时则将 [fp, #-20]、即 i[0] 放入 r0。

```

mov    r3, #12
str    r3, [fp, #-20]
mov    r3, #34
str    r3, [fp, #-16]
mov    r3, #56
str    r3, [fp, #-12]
mov    r3, #78
str    r3, [fp, #-8]
ldr    r3, [fp, #-20]
mov    r0, r3

```

利用栈机器实现函数调用的剩余过程请参见章节 3.2。

通过上文及章节 3.2 展示的事实可以看出，递归（如包含递归的排序算法）是函数的一种相对不好的使用方式，在假设没有优化条件的前提下，其缺点包含有：

1. 递归会大量浪费栈空间。

在递归执行过程中，没有一个被调用的函数过程执行到了 `return` 语句，函数全部没有返回，这造成了所有过程的栈帧空间全部没有释放。

2. 递归占用的内存难以观测。

函数内变量的数量可能是模糊的，配合条件分支语句便更加难以观测计算。此外，当前语言环境的变量栈和返回地址栈是否分开存放且其最大值、当前程序已经占用的栈等都是未知的因素。

3. 栈空间对浪费的容忍度低。

因为一般来说任何环境分配给程序的栈空间都不大，如 `node.js` 环境默认下分配给 JavaScript 运行程序的只有 984Kbyte 的容量，JVM 分配给 Java 语言的栈大小则一般默认为 512Kbyte。

利用 `wasm_arm` 平台可以验证这一认识。例如下方的递归代码，其极其简单，除了递归递增 `i` 变量 100000 次以外没有任何行为（使用 `arm gcc 8.2` 编译且优化参数为 `O3`）：

```

int num = 1;
int IAmRecursion(int i){
    int r = (i <= 100000) ? IAmRecursion(i + num) : i;
    if(r== -1) r++;
    return r;
}
int main(void){
    return IAmRecursion(0);
}

```

在 `wasm_arm` 平台中可以直观的看出，递归中的每一次函数调用，都产生了 8 字节的未被回收的栈空间消耗：

Memory																			\$\$\$
	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+a	+b	+c	+d	+e	+f		r00	00000032
001f6f00h	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00		r01	000f211c
001f6f10h	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00		r02	000e7744
001f6f20h	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00		r03	000e7744
001f6f30h	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00		r04	00000000
001f6f40h	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00		r05	00000000
001f6f50h	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00		r06	00000000
001f6f60h	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00		r07	00000000
001f6f70h	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00		r08	00000000
001f6f80h	00	00	00	00	90	6f	1f	00	40	29	00	00	98	6f	1f	00		r09	00000000
001f6f90h	40	29	00	00	a0	6f	1f	00	40	29	00	00	a8	6f	1f	00		r10	00000000
001f6fa0h	40	29	00	00	b0	6f	1f	00	40	29	00	00	b8	6f	1f	00		r11	001f6f90
001f6fb0h	40	29	00	00	c0	6f	1f	00	40	29	00	00	c8	6f	1f	00		r12	0000295c
001f6fc0h	40	29	00	00	d0	6f	1f	00	40	29	00	00	d8	6f	1f	00		r13(SP)	001f6f84
001f6fd0h	40	29	00	00	e0	6f	1f	00	40	29	00	00	e8	6f	1f	00		r14(LR)	00002940
001f6fe0h	40	29	00	00	f0	6f	1f	00	40	29	00	00	f8	6f	1f	00	r15(PC)	00002938	
001f6ff0h	40	29	00	00	00	70	1f	00	40	29	00	00	08	70	1f	00	CPSR <th>800000df</th>	800000df	
001f7050									r15		r13		View Location				SPSR <th>800000df</th>	800000df	

对比将编译后结果逆向得到的 arm 汇编（注意此刻优化参数为 O3）：

可以看出,在每次跳转至下一调用的递归过程中,指令 `push {fp,lr}` 确实占用了 8 字节的栈。此外通过观察剩余的指令,可知寄存器 `r0` 用于存放了不断加一过程中的中间变量 `r`,而 `r3` 寄存器用于存放变量 `num` 的地址及利用其获取变量 `num` 的值。

由于该平台分配了 1Mbyte 的栈空间，因此至多 131072 次调用便会将其全部填满。此外，递归过程中栈帧的不断开辟与递归结束后的栈帧的不断回退还会产生大量的多余时间消耗。

```
int num = 1;
int IAmRecursion(int i){
    return (i <= 100000) ? IAmRecursion(i + num) : i;
}
```

便可触发一般编译器的 tail-call optimization^[14]（即函数末尾的 return 处产生的函数调用没有开辟新栈帧的必要）。而利用 tail-call 进行递归则被称为 tail-end recursion，即尾递归。

由于函数末尾时刻的栈帧内容已实质上对于该函数内的指令没有用处，因此该函数栈帧可被 tail-call 的目标函数栈帧取代，没有开辟新栈帧的必要。不过，对于非递归情景而言，该优化可能仅能消除一次新栈帧的开辟，效果会十分有限。

其编译后结果逆向得到的 arm 汇编变为如下：

```
IAmRecursion(int):
    str    fp, [sp, #-4]!
    ldr    r3, .L8
    cmp    r0, r3
    add    fp, sp, #0
    bgt    .L1
    ldr    r2, .L8+4
    ldr    r2, [r2]
.L3:
    add    r0, r0, r2
    cmp    r0, r3
    ble    .L3
.L1:
    add    sp, fp, #0
    ldr    fp, [sp], #4
    bx     lr
.L8:
    .word  100000
    .word  1
```

可以看到，此处的 r0 依然用作中间变量，r2 寄存器用于存放变量 num 的地址及利用其获取变量 num 的值，不过没有了函数调用的过程，且在循环中也没有产生栈的增长。

总之可以验证，利用大量递归的算法（如利用递归排序）若不同时采用尾递归的方式并使用优化参数，空间和时间开销可能均会超出预想，因此应当尽量避免使用。此种递归的调用导致了一般的容量不大的栈的浪费，且任何递归逻辑都能通过改写为循环增强程序可读性并对 debug 过程友好。

4.2.2 从面向对象到处理器二进制指令

对象的生命周期维持完全与对象和类本身无关。对象的生命周期维持只有两种可能形态：直接栈上创建的形态和调用 `new()` 函数在堆上创建的形态。这分别对应了 C 语言中直接创建变量和利用 `malloc()` 函数申请内存存储变量两种方式，因此本篇不涵盖这一内容，这一内容请参见章节 4.3.1。

在 C++ 的类中，成员在堆上（或称为 free store 上，然而纠缠 free store 和堆的区别毫无意义，因为在标准 C++ 库中 free store 内存的管理本身调用了 `malloc()` 函数，实际管理逻辑与堆无异）的变量的初始化实质上是在类的构造函数中完成的，例如如下代码：

```
class A {
public:
    int data = 111;
    ~A(){ data = 111; }
};
```

经编译后逆向得出的二进制指令中可以看到，该 A 类生成了实质上内容完全一致的构造函数和析构函数，尽管在 C++ 的源代码中并没有明确写出构造函数。

此外还可以发现，在构造和析构函数的开始与结尾部分，代码经历的过程和普通的函数毫无区别，都是栈帧创建和销毁的过程（具体请见注释）。

A::~~A() [base object destructor]:

```
str    fp, [sp, #-4]! ;旧 fp 压栈
add    fp, sp, #0    ;sp 放入 fp
sub    sp, sp, #12    ;sp 留出 12 字节空间、即栈顶增长
str    r0, [fp, #-8]  ;r0 存入[fp-8]，r0 是传入的“某参数”
ldr    r3, [fp, #-8]  ;[fp-8]放入 r3，即某参数放入 r3
mov    r2, #111       ;#111 放入 r2
str    r2, [r3]        ;r2、即#111 放入[r3]、即利用某参数做指针
ldr    r3, [fp, #-8]  ;[fp-8]放入 r3，即某参数放入 r3
mov    r0, r3         ;r3 放入 r0，即返回了某参数
add    sp, fp, #0     ;恢复旧 sp
ldr    fp, [sp], #4   ;恢复旧 fp，即所谓函数返回的栈帧回退
bx     lr
```

A::A() [base object constructor]:

```
str    fp, [sp, #-4]!
add    fp, sp, #0
sub    sp, sp, #12
str    r0, [fp, #-8]
ldr    r3, [fp, #-8]
mov    r2, #111
str    r2, [r3]
ldr    r3, [fp, #-8]
mov    r0, r3
add    sp, fp, #0
ldr    fp, [sp], #4
bx     lr
```

其中 data 作为 A 类的成员变量，在构造和析构这种成员函数中，可以明显看到利用了 r0 这一参数寄存器传入了“某个参数”，并利用该参数进行间接寻址，可以对比如下函数：

```
void A(){
    int data = 111;
}
```

其编译后逆向的汇编代码为：

```
A():
    str    fp, [sp, #-4]!
    add    fp, sp, #0
    sub    sp, sp, #12
    mov    r3, #111
    str    r3, [fp, #-8] ;可以看到此处的#111 直接存入了[fp-8]
    nop
    add    sp, fp, #0
    ldr    fp, [sp], #4
    bx     lr
```

诚然，若是普通函数的成员变量，直接存在于栈帧上即可。但是，若是类的成员函数，其对成员变量的访问需要通过隐藏的“某参数”进行间接寻址，而这一参数则是该被实例化的对象的地址。

此处在做出一个夸张的对比的同时验证这一认识：如下文 B 类的 BFoo() 成员函数通过 this 访问成员变量，和一个独立的 AFoo() 函数则利用 A 结构体引用做参数访问 A 结构体的成员变量。二者经编译产生的 arm 指令完全一致。

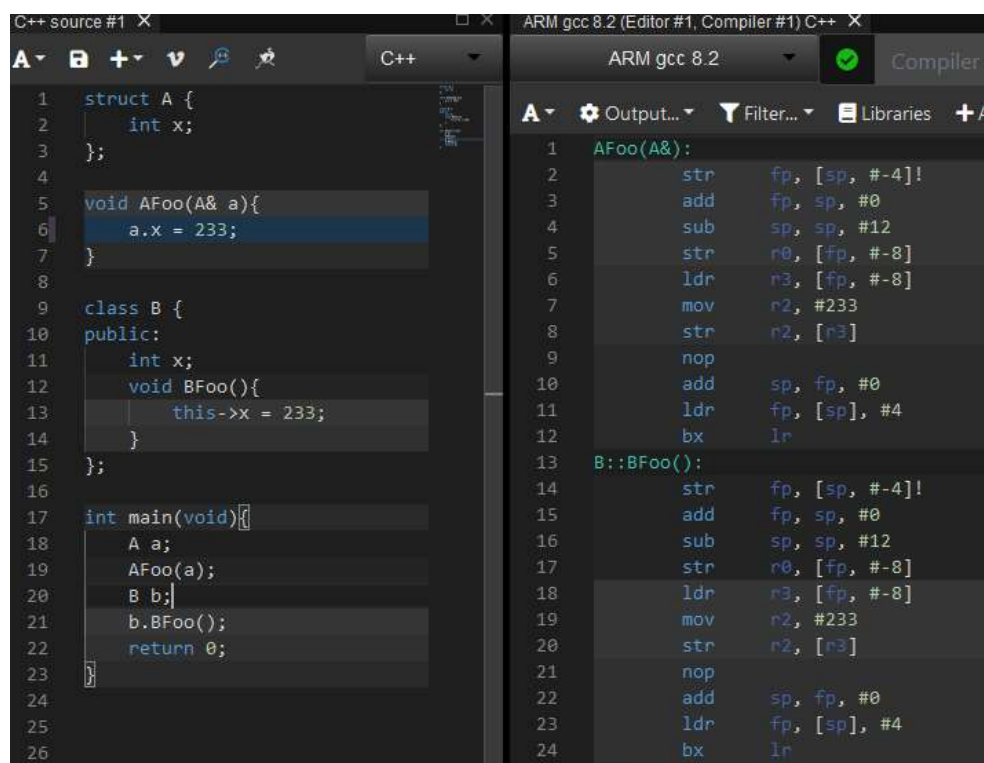


图 4-3 逆向后 AFoo 函数和 BFoo 函数对比

可以看到，在 C++ 语言中类的全部成员变量本身组成了一个 C 语言语境下的结构体，而成员函数对该结构体通过传入隐式的地址参数来访问该结构体。该隐式的地址参数在成员函数内通过 `this` 关键字访问。

至于 C++ 语境下的结构体，除了结构体默认访问权限是 `public` 以外，结构体即类，类即结构体，类本身不过是结构体的另一种表现形式。C 语言语境的结构体成为了 C++ 的结构体的子集。

此外，前文中提到是堆上的初始化过程存在于构造函数内，这是因为对栈上对象进行成员访问实际上没有运行期获取地址参数的必要——栈上的一切变量位置，乃至包含栈上的对象的成员变量的位置，在编译期就已经通过知晓其和函数的栈帧基址 `fp` 的偏移量从而提前确定了。

例如 C++ 代码：

```
class A{
public:
    int y = 111;
};

int main(void){
    A a;
    a.y = 2;
    return 0;
}
```

在编译后逆向得出的 arm 汇编代码为：

```
main:
    str    fp, [sp, #-4]!
    add    fp, sp, #0
    sub    sp, sp, #12
    mov    r3, #111
    str    r3, [fp, #-8] ;此处是 a 对象的初始化
    mov    r3, #2
    str    r3, [fp, #-8] ;而此处是成员变量赋值
    mov    r3, #0
    mov    r0, r3
    add    sp, fp, #0
    ldr    fp, [sp], #4
    bx     lr
```

可以看到，得益于栈本身的特质，`main()` 函数内 `a` 对象的成员变量 `y` 和函数内的普通变量没有区别，而 `a` 对象则变得隐形。其既并不存在、亦不需要构造函数来进行变量初始化。

至于类继承的效果在编译后是结构体的合并。尽管父类和子类成员变量名称可能相同，但在编译后依然是按照独立的两个变量看待，以应对面向对象中的子类→父类类型转换，例如如下示例：

```

class A{
public:
    bool x = true;
};

class B : public A {
public:
    bool x = false;
};

int main(void){
    B b;
    bool y = b.x;
    bool z = (static_cast<A>(b)).x;
    return 0;
}

```

其编译后逆向得到的 arm 指令为:

```

main:
    str    fp, [sp, #-4]!
    add    fp, sp, #0
    sub    sp, sp, #12
    mov    r3, #1
    strb    r3, [fp, #-8] ;可以看出[fp-8]是父类 A 的成员变量 x
    mov    r3, #0
    strb    r3, [fp, #-7] ;而[fp-7]是子类 B 的成员变量 x
    ldrb    r3, [fp, #-7]
    strb    r3, [fp, #-5] ;[fp-5]则是变量 y, 拷贝了子类 B 的 x
    ldrb    r3, [fp, #-8]
    strb    r3, [fp, #-6] ;[fp-6]则是变量 z, 拷贝了父类 A 的 x
    mov    r3, #0
    mov    r0, r3
    add    sp, fp, #0
    ldr    fp, [sp], #4
    bx     lr

```

通过以上示例不难看出, C++ 的类在成员函数、成员变量、构造析构函数, 类的继承中均隐藏了类的类型本身, 类的类型在编译后的代码中无处可寻。然而, 这种类型仅在编译期存在的设计在多态的实现中带来了一个问题: 由于没有类型的标记, 运行起无法安全处理父类→子类的类型转换。

例如如下代码:

```

A* a = new B();
B* b = dynamic_cast<B*>(a);

```

变量 a 的值可能会随运行期逻辑的变化而变化, 其可能变成 A 类的任意子类, 因而无法在编译期判断 a 到变量 b 的类型转换是否合法。

诚然, 此处可以直接进行强制类型转换, 但其转换结果是否正确就无法判断了。

总之，运行期的变量类型的存在是判断转换正确与否所必须的。例如一种设计是添加一个固定的 `const char[]` 存放于类及其子类中用以标记其名称，并在类型转换时用于校验。

而 `dynamic_cast` 实质上就是这一思想的一种实现方式。例如代码第二行编译后逆向得到的 `arm` 指令为：

```
ldr    r0, [fp, #-16] ; fp-16 为 a 变量的地址
cmp    r0, #0          ;
beq    .L14             ;其首先检查 a 变量是否为空指针
mov    r3, #0
ldr    r2, .L17
ldr    r1, .L17+4
bl     __dynamic_cast ;将 a 变量、.L17、.L17+4 传入 cast 函数
mov    r3, r0
b      .L15
.L14:
mov    r3, #0
.L15:
str    r3, [fp, #-20]
```

```
.L17:
.word  typeinfo for B ;而 L17 和 L17+4 就是 A 和 B 类的 typeinfo
.word  typeinfo for A
vtable for B:
.word  0
.word  typeinfo for B
.word  B::Foo()
vtable for A:
.word  0
.word  typeinfo for A
.word  A::Foo()
typeinfo for B:
.word  _ZTVN10__cxxabiv120__si_class_type_infoE+8
.word  typeinfo name for B
.word  typeinfo for A
typeinfo name for B:
.ascii "1B\000"
typeinfo for A:
.word  _ZTVN10__cxxabiv117__class_type_infoE+8
.word  typeinfo name for A
typeinfo name for A:
.ascii "1A\000"
```

可以看出，自动生成的 A 类的类型名称字符串为“1A”、B 类则为字符串“1B”。

4.3 WASM_ARM 微架构上简化的 C/C++标准库部分功能实现

4.3.1 内存管理

从底层来看，程序的数据无非在栈或堆上，然而栈和堆是十分不同的。栈单向增长，单向消减的特性，使其成为了一个能进行垃圾自回收的内存模型。其不仅有处理器的指令级支持（即一般存在压栈弹栈的相关指令），且由 C/C++语言的语法结构本身作为支柱（即函数的调用本身是栈帧的开辟，函数的返回是栈顶的回退）。

如在 C/C++中，一个函数内的全部局部变量均位于该函数被调用时所开辟的栈帧内，并通过栈帧基址的相对位置进行访问。这决定了：

1. 函数内代码访问局部变量时，无须提前确定局部变量的绝对地址，这使得函数能在任意位置进行任意次数的调用成为可能。
2. 一个函数在返回后，因为栈顶已经回退，如果此后的逻辑再次调用了任意函数，已返回的函数其中的变量可能变得不再可信（即脱离 scope）。
3. 因为如果在函数中调用其他函数被调用的其他函数必须确定其自身栈帧的起始位置（即当前函数栈的栈顶位置是必须确定的），所以一个函数内全部的变量长度总和必须是一个定值。

这同时也是为什么 C/C++中结构体其本身大小不可改变，因为结构体可以在栈上被定义。

而堆作为一个任意的内存区域，其管理方案取决于软件逻辑的具体实现。但一般来说，其存在的目的是为了解决栈的 2 和 3 这两个问题，以提供一个长期有效且长度可变的数据空间。堆的生命在程序执行期间永远存活，且其代表着可能申请到的数据空间的最大值。

那么，如果数据是长度可变：

1. 任何数据都永远无法一开始便直接预备其无限量的内存空间应对可能的增长，因为这使得下一数据无处存放。
2. 为了能有效利用空闲空间，数据长度减少时也应当立即将空间让出给其他数据使用。

即，单纯只是数据的长度增长/减少，就算没有进行任何读写操作，也应当是有代价的，其代价包含：

1. 空间管理代码执行的时间消耗。
2. 数据碎片化。如，数据增长但临近地址已被其他数据占有，因此只能从其他空闲位置获取（即形成了在线性空间中节点互相交错穿插的多个链表）。

若是不对指针进行封装，实质上由于地址的完全确定，无法对运行期正在存活的堆上数据进行碎片处理。这亦是 C++语言的一个特点。

在 wasm_arm 中提供了如下的一个简单的堆实现：

首先向 linker 标记一片区域 heap，由于 arm 架构特性，须确保其 4 字节对齐。

```
// malloc.h
constexpr size_t HEAP_SIZE = 0x00200000; //2MB
unsigned char __attribute__((section (".heap"))) heap[HEAP_SIZE];
// linker.ld
```



```
.heap : {
    __heap_start = . ;
    KEEP(*(.heap))
    . = ALIGN (4);
    __heap_end = . ;
} >RAM
```

在 wasm_arm 平台中，堆本身是一块由链表构成的内存区域，初始时仅有一个标记为未被使用的节点，且节点大小为整个堆内存区域，该结点的结构与初始化方式如下：

```
struct block_info {
    block_info *prev;
    block_info *next;
    size_t block_size;
    bool used;
} __attribute__((aligned (4)));
struct block_info *info_head = (block_info*)heap;

void infoHeadInit(){
    info_head->prev = nullptr;
    info_head->next = nullptr;
    info_head->block_size = HEAP_SIZE;
    info_head->used = false;
}
```

在程序申请内存时，malloc 函数首先找到未被使用且大小大于等于所请求的大小的节点，如该节点不存在则返回空指针。

如符合该条件的节点存在，若其大小大于所请求的容量与节点信息结构体的和，则将该节点拆分为两个子节点：其中一个占有代码请求的容量并将其置为已使用，另一个占有剩余空间并将其置为未使用。

在程序调用 free 函数请求释放时，则尝试将该节点标记为未使用并递归将当前节点和两侧未被使用的节点合并。

对于 gcc 编译器可使用 -Map, filename.map 链接器参数可以生成各个段/函数等标记在二进制文件中的位置。一般来说，特别是操作系统下的普通应用程序，当程序加载运行后这些地址会产生变化。然而在 wasm_arm 平台，由于采用单一平面内存执行单一程序，没有地址虚拟化等影响，因此该 map 的标记和实际内存地址一一对应。

4.3.2 智能指针

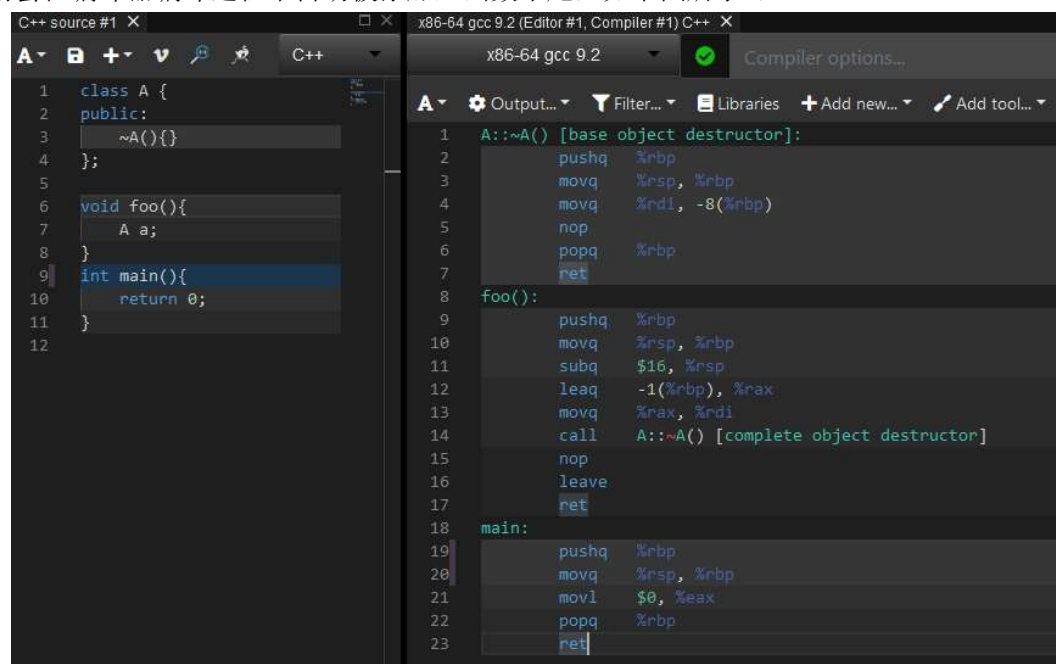
C/C++的指针本身是带有指向的值的类型的地址变量，其中类型辅助程序员正确使用该地址并在编译时警告程序员错误的使用逻辑；而地址则能指向任意地址空间中的位置。但一般来说，栈上的变量并不需要利用指针，因为栈上的变量将于函数返回时自动被回收。由于堆上申请到的空间不手动释放会导致其直到程序结束才会被回收，因此指针一般用于指向需要程序员进行内存回收的堆空间。

当指向堆上空间的指针在多处被使用，其理论上最万用且保险的策略是应在最后一次被使用后被回收。在C++的语言设计中，最重要的目标特性之一便是将一切可能在编译期实现的任务放于编译期完成。然而对于一个被多处共享的指针，不仅程序员往往难以判断何处才是最后一次的使用，加上代码执行的逻辑流程可能被运行期产生的状态控制（事实上，一段优秀的代码逻辑流程本身就应当完全被运行期产生的状态控制，因为编译期就能确定的状态所控制的逻辑流程都可以采用类似宏或 `constexpr if` 的编译期手段进行优化），即编译期编译器也亦无法利用过程调用树提前对指针使用情况进行追踪。因此，一个在运行期间追踪指针的使用情况的工具是必要的。

C++的智能指针的目的则是通过在运行期追踪指针的使用情况，在合适的时机释放资源，从而简化堆上指针管理的空间的资源释放部分。

通过在资源获取时便初始化包含对资源释放功能的生命周期的自动管理工具，C++的RAII（资源获取即初始化）法则确定了堆上资源获取及释放的一般模式，遵守该法则申请及释放堆空间可以有效避免内存泄漏并安全地进行资源的多 scope 共享。

此外，虽然逻辑流程在编译时不确定，但在C++中栈上的资源的生命周期在编译期是确定的，即函数返回时会触发该函数栈帧的释放。而对象被释放时又会触发该对象的析构函数（即，栈上的对象将在函数返回时被调用其析构函数也亦是在编译期确定的，这一调用会在编译器编译过程中自动被添加至函数末尾，如下图所示），



```
C++ source #1
1 class A {
2 public:
3     ~A(){}
4 };
5
6 void foo(){
7     A a;
8 }
9 int main(){
10     return 0;
11 }
12

x86-64 gcc 9.2 (Editor #1, Compiler #1) C++
x86-64 gcc 9.2
Compiler options...

Output... Filter... Libraries + Add new... Add tool...

1 A::~~A() [base object destructor]:
2     pushq %rbp
3     movq %rsp, %rbp
4     movq %rdi, -8(%rbp)
5     nop
6     popq %rbp
7     ret
8
9 foo():
10    pushq %rbp
11    movq %rsp, %rbp
12    subq $16, %rsp
13    leaq -1(%rbp), %rax
14    movq %rax, %rdi
15    call A::~~A() [complete object destructor]
16    nop
17    leave
18    ret
19
20 main:
21    pushq %rbp
22    movq %rsp, %rbp
23    movl $0, %eax
24    popq %rbp
25    ret
```

图 4-4 在逆向后二进制指令中所存在的被编译器自动添加的析构函数调用

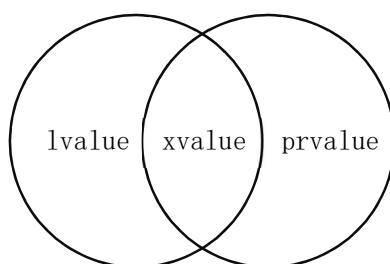
因此，可以利用一个位于栈上的对象，通过其构造及析构函数被调用的情况追踪资源被利用的过程。这便是智能指针最本源的核心思想。

在 C++98 中，智能指针 `auto_ptr` 首次被添加进入标准库，其功能为确保一个指针在传入 `auto_ptr` 对象后在该对象脱离 scope 时释放该指针的内存空间，同时仅有一个 `auto_ptr` 维护该指针。

该智能指针的实现及其简单：通过构造函数参数或赋值符号传入指针时持有新的指针，传入其它 `auto_ptr` 对象的引用时则将该对象内的指针转移至自身并将原对象持有指针置为空；在析构函数被调用或赋值传入新指针时则释放已持有的指针。

然而，该智能指针包含许多问题，其中最为严重的一个问题甚至动摇了 C++ 的根本的语法规则。即新的 `auto_ptr` 对象取代原 `auto_ptr` 对象维护的指针时须传入原对象的引用并置空原对象持有的指针，这一行为本质实际上是通过 copy syntax（因赋值的等于号本质上是对目标的复制行为）进行的移动语义（即移动了原资源的控制权限）的实现（尽管移动语义在 C++11 中才被提出，并且该实现方法此后被认为是错误的）^[15]。限于其语法规则的时代背景，C++98 和 C++03 在使用中并没有细化左右值的引用区分（其实左右值在 C++98 的标准中就有了^[16]，只是 lvalue 和 rvalue 二者间没有交集，即没有 xvalue、prvalue、glvalue，且在 C++98 和 C++03 标准原文中提到“Every expression is either an lvalue or an rvalue”^[16, 17]），更没有 `std::move()` 等标准库函数将左值转换为右值（即将 lvalue 转为 glvalue 和 rvalue 的交集 xvalue）。赋值符号和构造函数被重载用于传入对其它 `auto_ptr` 的引用同时占据了左值引用和右值引用。这导致了 `auto_ptr` 对象无法通过赋值符号被复制，这种对赋值语法的利用方式产生了同现有标准库内及普遍使用的算法的不兼容^[15, 18]。同时在重载的赋值符号内修改了原值这一行为也引发了普通 C++ 开发者对赋值符号的功能权限的恐慌。

为了修正 `auto_ptr` 的问题，C++11 标准不仅带来了新的左右值区分引用和其交集作为现行的补充^[12]，并且还在标准库里提供了 `std::move(v)`（用于将 lvalue 转换为 xvalue）、`std::forward<T>(t)`（用于将 lvalue 还原为原先的 rvalue，因为一般函数参数 rvalue 引用传入后作为有 identity 的变量就变成了 lvalue，因此可应用于 forward 右值引用的参数的右值状态给调用的函数中，例如章节 3.7.1 的代码示例）等编译期函数对左右值状态进行操作，同时也带来了 `unique_ptr`，`shared_ptr` 和 `weak_ptr` 三个新的智能指针类型。



Has Identity: $glvalue = lvalue \cup xvalue$

Movable: $rvalue = xvalue \cup prvalue$

图 4-5 lvalue、xvalue 和 prvalue 的关系

`Unique_ptr` 在功能上和 `auto_ptr` 几乎相同：仅限一个 `unique_ptr` 指向资源，在这唯一一个 `unique_ptr` 脱离 scope 或主动释放的情况下释放资源。其目的是作为原 `auto_ptr` 的功能替代品，然而其删除了拷贝构造函数和拷贝赋值函数，传入其它的 `unique_ptr` 引用仅能通过右值引用传入赋值符号中，即使用赋值符号须强制调用 `std::move()` 函数。这使得单纯拷贝赋值和赋值并修改原值两个行为得以被区分，解决了恐慌。

此外，`shared_ptr` 和 `unique_ptr` 在使用中最大的区别是其在传入另一 `shared_ptr` 时

既可以选择剥夺对方的维护权, 即利用 `std::move()` 转为右值引用 (准确来说是 `xvalue` 引用, 因为依然含有 `identity`)。此外这将使得 `shared_ptr` 与 `unique_ptr` 没有太大区别, 引用计数也亦不会自增), 亦可以选择左值传入保留另一 `shared_ptr` 的维护权。每一个维护相同指针的 `shared_ptr` 对象都拥有同一个针对该指针的引用计数器, 在左值传入 `shared_ptr` 时会+1 引用, 析构函数被调用时则会-1 引用。引用变为 0 时则会释放指针空间。这样只有当所有 `shared_ptr` 脱离 `scope` 或主动释放的情况下才会释放资源。其与 `unique_ptr` 相比下额外支持了所维护资源的跨过程分享。

至于 `shared_ptr` 的代码实现可以秉持开放思维进行设计^[19]。其一般的设计包含三种并各有优劣:

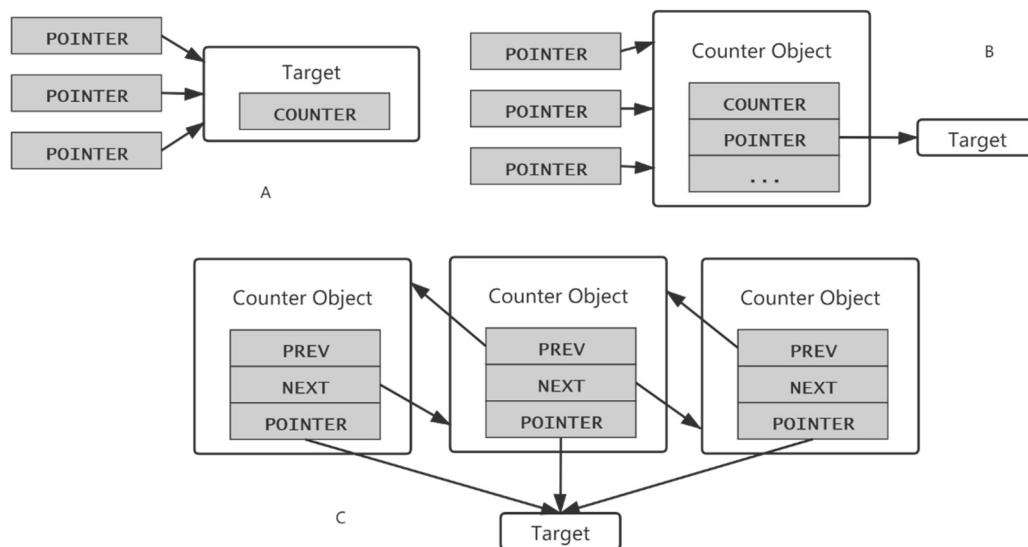


图 4-5 几种 `shared_ptr` 的实现方式

在 C++ 标准库中采用了类似图 B 的实现, 并且其在 `counter object` 中还包含了一个在构造函数中传入的 `type-erased deleter`, 以方便在维护同一类型的不同对象时采用不同的 `deleter` (`unique_ptr` 中碍于性能考虑, 没有实现 `type-erased deleter`), 其实现如:

```
std::function<void (void*)> deleter;
```

如果两个同一 `scope` 内的 `shared_ptr` 所指向的资源内互相引用对方, 在二者脱离 `scope` 后二者的引用计数均会减 1, 并不会任何一方归 0, 这导致了二者均不会被回收。

若只是一侧单向引用另一侧的情况, 尽管脱离 `scope` 时管理 A 侧的 `shared_ptr` 的析构函数被触发时 B 侧可能依然保有对 A 侧的引用, A 本身的析构函数此时不会被调用亦不会被回收, 但由于 A 的 `shared_ptr` 的析构函数已经触发, 对 A 的引用已经减 1。此后在 B 的 `shared_ptr` 的析构函数触发时由于 B 未被其它资源引用, B 被回收。这导致了再度触发 A 的 `shared_ptr` 的析构函数, 从而使对 A 的引用归零, 此时便会回收 A 并触发 A 的析构函数。

这一问题的关键点是, 若 B 又被 A 所引用, B 则不会被回收, 从而不会产生单侧引用时后半段的行为。

为了触发后半段的行为, A 或 B 其中一方的 `shared_ptr` 可以替换为 `weak_ptr`, 其可指向 `shared_ptr` 所指向的资源, 不过 `weak_ptr` 的构造和析构函数的触发并不会改变资源的引用计数器。其没有重载 `->` 和 `*` 运算符因此不可直接访问到资源 (当然不可以), 但可通过其 `lock()` 成员函数获得资源, 资源有效时将返回一个 `shared_ptr`, 在资源无效时返回空。

因此 weak_ptr 亦可用于检查资源有效性。

总而言之，智能指针本身由 C++ 实现，其本身不是语法规则。但其功能的健全实现依赖于新的语法规则。其本身的特性也来自于 C++ 的“尽可能将任务放在编译期”和“实现更多功能”这两个 C++ 语言设计目标间的妥协。

智能指针的具体实现可参考 gcc 编译器源码，在 10.2.0 版本中的实现分别位于路径：

```
gcc-master\libstdc++-v3\include\backward\auto_ptr.h
gcc-master\libstdc++-v3\include\bits\unique_ptr.h
gcc-master\libstdc++-v3\include\bits\shared_ptr.h
```

（其中 weak_ptr 的实现在 shared_ptr.h 内）

4.3.3 输出接口

本平台采用了一个极简的输出，该输出由 wasm_arm 环境内逻辑将对应地址空间的内容交与 Js 环境，并调用浏览器 DOM 渲染。

```
constexpr std::size_t SCREEN      = 0x01FFFF00;
constexpr std::size_t SCREEN_SIZE = 0x00000100;
std::string CPUSimGetScreen(){
    std::string rtn = "";
    for(std::size_t addr = SCREEN;
        addr < (SCREEN + SCREEN_SIZE);
        addr++)
    ){
        rtn += static_cast<char>(arm7sim->GetMemory8(addr));
    }
    return rtn;
}
emscripten::function("cpuSimGetScreen" , &CPUSimGetScreen);
```

```
function updateViewScreen(){
    $("#screen").val('');
    var str =
        window.Module.cpuSimGetScreen()
            .replace(/[\x00-\x7F]/g, ' ')
            .replace(/[\n\r]/g, ' ');
    for(var i=0; i<8; i++){
        var sub = str.substr((i*32), 32);
        $("#screen").val(
            $("#screen").val() + sub + ((i==7) ? "" : "\n")
        );
    }
}
```

4.4 ARM 指令集架构模拟内的 ARM 指令集架构模拟

由于 wasm_arm 平台在开发过程中避免了对运行平台的依赖，其代码本身便可以针对多平台编译，因此亦可将利用该 arm 模拟器开发的 wasm_arm 平台自身作为该 arm 模拟器的编译目标。

例如下方代码在 RESET 中断向量处放置了三枚简单的指令执行，并在每条指令执行完毕后将结果输出至 wasm_arm 的基本输出设备。

```
ARM7TDMI_DEBUG<1024> *cpusim = new ARM7TDMI_DEBUG<1024>();
cpusim->SetMemory32(0x0,{
    0x0100A0E3,//mov r0, #1
    0x010080E2,//add r0, #1
    0x010080E2,//add r0, #1
});
cpusim->cpu.EXCEP_RST();

print("0x"); print(cpusim->cpu.instruction_register, 16); print(" : ");
cpusim->cpu.Step();
print("r0 is "); print(cpusim->cpu.registers[0]); print("\n");

print("0x"); print(cpusim->cpu.instruction_register, 16); print(" : ");
cpusim->cpu.Step();
print("r0 is "); print(cpusim->cpu.registers[0]); print("\n");

print("0x"); print(cpusim->cpu.instruction_register, 16); print(" : ");
cpusim->cpu.Step();
print("r0 is "); print(cpusim->cpu.registers[0]); print("\n");
```

其运行结果如下：



图 4-6 模拟器中执行模拟器模拟的输出

5 总结

计算机学科作为市场性极强的一门技术性工程学科，无论从高度抽象化的面向具体应用的角度抑或是底层的面向芯片的角度来看，随着时代的发展，不仅技术工具的多样性越发五花八门，认知的层次也亦有着不断的变化。但所有的一切变化都有着一个同样的趋势：许多以往手动的事情变得自动了，以往暴露的事物被封装变得抽象了，于是一切都变得看起来简单了。

以一个激进的新 C++ 标准提案 proposal 1308 为例^[20]：

```
void takeDamage(player &p) {  
    inspect(p) {  
        [hitpoints: 0, lives:0] => gameOver();  
        [hitpoints:hp@0, lives:1] => hp=10, l--;  
        [hitpoints:hp] if (hp <= 3) => { hp--; messageAlmostDead(); }  
        [hitpoints:hp] => hp--;  
    }  
}
```

图 5-1 Pattern Matching

C++简直是要变成 python 了！

随着时代的发展，事物都有着变成一个高度复杂且易于使用的“黑箱”的趋势，“黑箱”也会变得越来越大，使用者也会离“黑箱的核心”越来越远。现代的一般项目开发的重要规则之一是各部分独立封装、以实现分工协作的同时亦可以对“黑箱”内部不甚了解，借此应对这一生态的多样性和复杂性，达到管理上的效率最优化。然而，这一做法又会反过来加强这一“黑箱”特质。

而 wasm_arm 平台的目的是反过来消除计算机内各个方向与部分的独立性，打破这一“黑箱”，并尝试将一切事物拆开来研究并试图回归本源，并以此打通浏览器、Javascript、C++、汇编、处理器、乃至 1 与 0 之间的联系。

Wasm_arm 平台及其衍生项目以实现浏览器内的 arm 架构模拟器作为切入点，自下而上，从二进制指令的 1 与 0 逐层递进，其目的在于做出一个简单但较为完整的体系，并尝试跨越模块间的隔阂：语言和语言间的，软件和硬件间的，沙箱内与沙箱外的，底层开发考虑的与高层开发考虑的。而本文则是对 wasm_arm 平台及其开发过程中的一些事物进行的一番大致的总结。

本文从 arm 处理器的特性和指令集出发，引用 x86_64 和 MIPS 架构做了一些对比，讨论了其特点及对其的实现，同时提到了一个较为有趣的准确性验证方案，借此通过 arm 二进制接口部分展示了 C++特性和汇编的关系，同时借此契机提到 C++的一些新标准新功能和一些旧标准。

本文尽量避开基础知识的说明，更多关注于完成这一体系实际遇到的具体问题及其思维方式的系统结构，延伸出对软件与硬件结合处的思考，并旁征博引，对以往所接触到的其他事物做了一些对比。希望能起到一个对多个具体事物产生整体理解并能统括全局的作用。

同时，本平台利用了新的 WebAssembly 技术，展示了 WEB 前端未来的发展潜力。

附录

ARM 架构 C++编译器、版本、参数:

```
ARM gcc 8.2 -O0 -march=armv4t
```

32 位 x86_64 架构 C++编译器、版本、参数:

```
x86-64 gcc 9.2 -m32 -O0 -std=c++14
```

64 位 x86_64 架构 C++编译器、版本、参数:

```
x86-64 gcc 9.2 -m64 -O0 -std=c++14
```

Emscripten C++编译器参数:

```
emscripten -O3 -std=c++2a -s STACK_OVERFLOW_CHECK=0 -s  
SAFE_HEAP=0 -s VERBOSE=0 -s EXIT_RUNTIME=0 -s  
DISABLE_EXCEPTION_CATCHING=1 -s AGGRESSIVE_VARIABLE_ELIMINATION=1  
-s WASM=1 -fno-exceptions -flto -fno-rtti -  
DEMSCRIPTEN_HAS_UNBOUND_TYPE_NAMES=0 -s ALLOW_MEMORY_GROWTH=0 -s  
INITIAL_MEMORY=134217728 -s ASSERTIONS=0 -s RUNTIME_LOGGING=0
```

由于 wasm_arm 平台没有动态内存的需要，因此配置了 128MB 的不可增长内存。

参考文献

- [1] WebAssembly Community Group, A. Rossberg (editor).
WebAssembly Specification Release 1.1 [EB/OL].
https://webassembly.github.io/spec/core/_download/WebAssembly.pdf
- [2] Intel Corporation.
Intel® 64 and IA-32 Architectures Software Developer's Manual [EB/OL].
<https://software.intel.com/content/dam/develop/external/us/en/documents-tps/325462-sdm-vol-1-2abcd-3abcd.pdf>
- [3] ARM Limited.
ARM7TDMI Technical Reference Manual [EB/OL].
<https://developer.arm.com/documentation/ddi0210/c/>
- [4] Integrated Device Technology, Inc.
IDT R30xx Family Software Reference Manual [EB/OL].
<https://cgi.cse.unsw.edu.au/~cs3231/doc/R3000.pdf>
- [5] L. Maranget, S. Sarkar, P. Sewell.
A Tutorial Introduction to the ARM and POWER Relaxed Memory Models [R].
University of Cambridge, 2012.
<https://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf>
- [6] A. Fog.
Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD, and VIA CPUs [R].
Technical University of Denmark, 2021.
https://www.agner.org/optimize/instruction_tables.pdf
- [7] ARM Limited.
Procedure Call Standard for the ARM Architecture [EB/OL].
<https://web.eecs.umich.edu/~prabal/teaching/resources/eecs373/ARM-AAPCS-EABI-v2.08.pdf>
- [8] A. Fox.
A HOL Specification of the ARM Instruction Set Architecture [R].
University of Cambridge Computer Laboratory, ISSN1476-2986, 2001.
<https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-545.pdf>
- [9] V. Pfau.
Classic NES Series Anti-Emulation Measures [EB/OL].
<https://mgba.io/2014/12/28/classic-nes/>

- [10] A. Fox.
Verifying the ARM Block Data Transfer Instructions [R].
University of Cambridge, 2004.
<https://acjf3.github.io/papers/bdt.pdf>
- [11] V. Pfau.
Cycle Counting, Memory Stalls, Prefetch and Other Pitfalls [EB/OL].
<https://mgba.io/2015/06/27/cycle-counting-prefetch/>
- [12] ISO/IEC 14882:2011.
Information technology — Programming languages — C++ [S].
https://www.techstreet.com/standards/incits-iso-iec-14882-2011-2012?product_id=1852925
- [13] ARM Limited.
Exception Handling ABI for the Arm® Architecture [EB/OL].
<https://github.com/ARM-software/abi-aa/releases/download/2020Q4/ehabi32.pdf>
- [14] A. Krall, U. Neumerkel.
Proper Tail Recursion in C [R].
Vienna University of Technology, 2001.
<https://www.complang.tuwien.ac.at/schani/diplarb.ps>
- [15] Open STD Org.
Rvalue Reference Recommendations for Chapter 20 [EB/OL].
<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1856.html>
- [16] ISO/IEC 14882:1998.
Programming languages – C++ [S].
<https://shop.standards.govt.nz/catalog/14882%3A1998%28ISO%7CIEC%29/view>
- [17] ISO/IEC 14882:2003.
Programming languages – C++ [S].
<https://infostore.saiglobal.com/store/details.aspx?ProductID=712174>
- [18] H. Sutter.
The New C++: Smart(er) Pointers [EB/OL].
<https://www.drdobbs.com/cpp/the-new-csmarter-pointers/184403837>
- [19] Y. Sharon.
Smart Pointers – What, Why, Which? [EB/OL].
<http://ootips.org/yonat/4dev/smart-pointers.html>

[20] Open STD Org.
Pattern Matching [EB/OL].
<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1308r0.html>