# 实验二（硬件插桩与分支预测模拟器选题）说明文档

## ○ 声明

请不要把此文档和实验代码上传到互联网，感谢各位同学的配合！

实验代码以及说明文档可能存在一些笔误或者表述上的问题，请选择本实验的同学及时给出反馈，可以根据反馈情况获得一定加分。

## 一 实验介绍

### 1 实验信息

本次实验名为“ARM 处理器的指令插桩与分支预测器模拟实验”，实验内容分成两大部分，一是基于 ARM 的 CoreSight 调试架构实现简单的片上跨核调试与指令插桩工具，二是在这个工具的基础上完成一个分支预测器的模拟。在第一部分中，工具的框架和大部分实现细节已提供，要求同学们根据本文档和代码提示完成部分函数的实现。在第二部分中，要求同学们根据“计算机系统结构”课程中的 Branch Prediction 相关内容，模拟一个简单的分支预测器。

本次实验的设备统一为树莓派 3B+。树莓派 3B+使用了 4 个 ARM Cortex-A53 处理器，但出于低功耗考虑，也对处理器的设计进行了一定的修改[1]，比如裁剪了一些预测执行技术。Cortex-A53 处理器属于 ARM 处理器的 v8 版本，能够支持 AArch32 和 AArch64 两种指令架构。本次实验使用 AArch64 指令架构。

树莓派的官方操作系统 Raspbian 主要提供了 32 位系统的支持，然而，本次实验需要使用 64 位系统。因此，为了更好地支持 64 位系统的实验环境，本次实验使用 Ubuntu 20.04 的 64 位版本[2]。

文档接下来的章节包括：第一章的实验目的、实验要求，第二章的背景知识，第三章的实验内容，第四章的评分标准，第五章的附录（关于树莓派配置），以及第六章的参考资料。

### 2 实验目的

本次实验属于创新实验，在课程内容的基础上有所提高。实验的代码量较少，但需要同学们花费一定的课外时间阅读文档以及查阅一些额外的资料。实验中涉及到的 ARM 指令集、ARM 调试架构，均是课程中未介绍的，本文档将会给出概要性的介绍。本次实验也需要对 Linux 系统编程进行初步学习，了解简单的内核模块开发方法。

通过本次实验，同学们可以有以下收获：

- 进一步熟悉分支预测器的设计

- 初步了解 ARM 64 位指令集 A64
- 初步了解 ARM 调试架构
- 初步学习 Linux 内核驱动开发

## 3 实验要求

实验采取线下检查 + 线上提交实验报告的模式。

本次实验日期为**第 13 周至第 17 周**，之后算作补交。最晚**第 18 周周四**前提交实验报告并归还实验设备，否则实验不得分。

如第 1 小节所述，本次实验分成两大部分，第一部分是指令插桩工具的完善，第二部分是分支预测器的模拟。指令插桩工具用于对测试用例进行指令粒度的跟踪，一旦跟踪到分支指令，就调用同学们实现的分支预测算法，预测分支的目标地址，并与计算出的实际分支地址比较，记录正确率。

第一部分分成两个子任务 task1 和 task2。task1 要求同学们找到树莓派 3B+的调试寄存器组的物理基地址，需要完成一个简单的内核模块并通过测试。task2 要求同学们实现跨核调试过程中调试主机（debug host）和调试目标（debug target）的两种数据传输模式，需要实现内核模块的三个相关函数并通过测试。总体代码量不超过 50 行。

第二部分分成两个子任务 task3 和 task4。task3 要求同学们用 C 语言模拟一个分支预测器，在输入一个分支指令的 PC 时返回该分支指令跳转的目标地址。task4 要求同学们给出一个测试用例，最终测试正确率时会执行所有同学的测试用例。该部分没有代码量限制，但分支预测器的空间规模受限，**至多为 1KiB**。

# 二 背景知识

## 1 A64 指令集

### 1.1 A64 指令集简介

在 AArch64 架构下执行的指令集被称为 A64 指令集，这是一种 RISC 指令集，所有的 A64 指令都是 32 位长度。A64 指令可以分成以下几种类别：

- 控制流相关指令和系统指令：包括分支指令、异常触发指令和系统指令
- 通用寄存器运算指令：根据操作数的类型，分成带立即数的指令和不带立即数的运算指令
- 访存指令：包括各种 Load 和 Store 指令
- 向量和浮点运算指令：以向量和浮点寄存器为操作数的指令

本次实验主要涉及分支指令和部分系统指令。

A64 指令集的操作数域指定了指令的操作数。如果操作数是通用寄存器，那么只使用 5 位的编码；

如果操作数是立即数，那么会使用不定长度的编码，并且可能会在硬件层面对编码进行一定的移位或组合。

## 1.2 通用寄存器

下面对 AArch64 的通用寄存器进行简单介绍。

上述的操作数域用 5 位编码表示通用寄存器，能够表示 32 个通用寄存器。实际上，有 31 个通用寄存器和 2 个特殊寄存器能够在这个编码表示。其中，通用寄存器被命名为 R0-R30，A64 指令集提供了灵活使用通用寄存器的方式。如果只访问这些寄存器的低 32 位，那么就把这些寄存器用别名 W0-W30 代替。如果访问这些寄存器的全部 64 位，那么就把这些寄存器用别名 X0-X30 代替。特别地，读 W0-W30 时会忽略寄存器的高 32 位，写 W0-W30 时会把寄存器的高 32 位置 0。需要注意的是，A64 指令集不存在 W31 或 X31 寄存器。

2 个特殊寄存器分别为零寄存器 ZR 和栈指针寄存器 SP。零寄存器的内容恒等于零。栈指针寄存器的内容为当前进程的函数调用栈的栈顶，用于在异常处理等跨域操作结束后，重新回到当前进程的上下文环境。这两个寄存器都支持 32 位和 64 位的访问模式。32 位模式下两个寄存器的别名分别为 WZR 和 WSP；64 位模式下两个寄存器的别名分别为 XZR 和 SP。这两个寄存器在指令内的操作数编码均为 31，在不同指令类型下由硬件解释。表 1-1 总结了上述寄存器的属性。

表 1-1 通用寄存器的相关属性

| Name | Size | Encoding | Description |
|---|---|---|---|
| Wn | 32 bits | 0-30 | General-purpose register 0-30 |
| Xn | 64 bits | 0-30 | General-purpose register 0-30 |
| WZR | 32 bits | 31 | Zero register |
| XZR | 64 bits | 31 | Zero register |
| WSP | 32 bits | 31 | Current Stack Pointer |
| SP | 64 bits | 31 | Current Stack Pointer |

需要注意的是，AArch64 不显式提供 PC 寄存器，但可以通过 ADRP[3]和 ADR 指令计算出当前时刻 PC 与某个 label 的偏移量，从而间接地得到 PC 值。比如，指令 “ADR X0, .” 能够把该指令的 PC 值写入通用寄存器 X0。此外，对于进入调试状态的处理器，进入调试状态之前的最后一个指令地址会被存在一个系统寄存器 DLR_EL0 内，调试器可以通过这个指令得到进入调试状态前的 PC 值。

## 1.3 寻址方式

AArch64 架构的虚拟地址长度为 64 位。其中，用户空间虚拟地址的高 63-48 位为 0，内核虚拟地址则都是 1。比如，0x0000aaaaaaaaa800 是 Linux 系统未启用 ASLR 机制时一个典型的用户空间虚拟地址，而 0xffffffffffff100 则是一个典型的内核空间虚拟地址。

AArch64 主要有五种寻址方式：

- Base register mode：把寄存器内的值作为虚拟地址，表示为：[base{, #0}]

- Base plus offset mode：寄存器内的基地址加上寄存器/寄存器移位/立即数的偏移结果作为虚拟地址，表示为：① [base{, #imm} ② [base, Xm{, LSL #imm}] ③ [base, Wm, (S|U)XTW {#imm}]
- Pre-indexed mode：把基地址+偏移作为虚拟地址，然后把该虚拟地址写入存有基地址的寄存器，表示为：[base{, #imm}]!
- Post-indexed mode：把基地址作为虚拟地址，然后把该基地址+偏移作为新的虚拟地址写入存有基地址的寄存器，表示为：① [base], #imm ② [base], Xm
- Literal(PC-relative) mode：以 PC 加上某个偏移量为虚拟地址。表示为：<label>

本次实验中，需要给出分支预测器的预测地址。为了减小分支预测器的空间开销，可以结合用户空间虚拟地址的特点，以及各种寻址方式的区别，用合适的方式存储并计算预测地址。

## 1.4 条件标志与条件码

与 X86 架构的处理器类似，AArch64 架构的处理器也实现了条件标志（condition flag）。处理器提供了 4 个条件标志，分别为 N，Z，C 和 V，它们与 X86 指令集的含义相同。ARM 可以通过 CMP 等指令显式地设置条件标志。

A64 指令集通过条件码（condition code）来使用条件标志，比如条件分支指令“B.ne <label>”中，如果条件标志中 Z 等于 0，指令就跳转到<label>位置执行。A64 的条件标志如表 1-2 所示。

表 1-2 A64 条件码

| cond | mnemonic | meaning | condition |
|---|---|---|---|
| 0000 | EQ | Equal | Z == 1 |
| 0001 | NE | Not equal | Z == 0 |
| 0010 | CS or HS | Carry set | C == 1 |
| 0011 | CC or LO | Carry clear | C == 0 |
| 0100 | MI | Minus, negative | N == 1 |
| 0101 | PL | Plus, positive or zero | N == 0 |
| 0110 | VS | Overflow | V == 1 |
| 0111 | VC | No overflow | V == 0 |
| 1000 | HI | Unsigned higher | C == 1 && Z == 0 |
| 1001 | LS | Unsigned lower of same | !(C == 1 && Z == 0) |
| 1010 | GE | Signed greater than or equal | N == V |
| 1011 | LT | Signed less than | N != V |
| 1100 | GT | Signed greater than | Z == 0 && N == V |
| 1101 | LE | Signed less than or equal | !(Z == 0 && N == V) |

| | | | |
|---|---|---|---|
| 1110 | AL | Always | Any |
| 1111 | NV | Always | Any |

## 1.5 分支指令

分支指令是 A64 指令集的重要组成部分，也是本次实验中需要用到的指令类型，因此，本节对 A64 中的分支指令进行详细介绍。

如果按是否必然跳转来分类，A64 指令集中的分支指令可分为条件分支和非条件分支；如果按寻址方式来分类，A64 指令集中的分支指令可分为直接跳转分支和间接跳转分支。其中，间接跳转分支均为非条件分支。

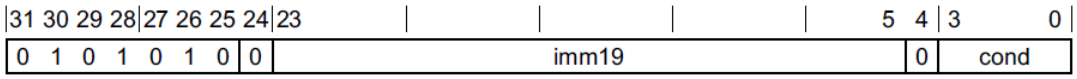
A64 中的条件分支包括五种分支指令，分别是：B.cond，CBNZ，CBZ，TBNZ 以及 TBZ。非条件分支包括 B，BL，BLR，BR 和 RET 指令，以及特殊的 ERET 指令。

### 1.5.1 B.cond

B.cond 是最常用的条件分支指令，直接根据当前条件码判断是否跳转，拥有 2MB 的寻址空间。该指令通常与条件码的置位指令，如 CMP 指令一起使用。

指令的汇编格式为“B.<cond> <label>”。其中，cond 表示条件码，label 表示条件为真时的跳转地址。

指令的编码如下：



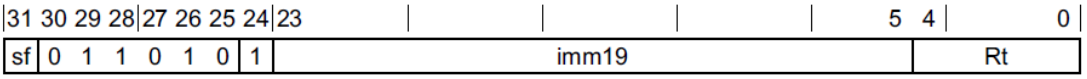

其中，cond 表示 4 位的条件码，imm19 用于表征跳转目标<label>与这条分支指令地址的偏移量。因此，该指令用到了 PC 相对寻址的模式。该偏移量 offset 的计算方式为，imm19 左移两位并且带符号扩展到 64 位。而最终跳转地址的计算方式为：<label> = pc + offset。

### 1.5.2 CBNZ

CBNZ 是先比较，后跳转的指令，需要取某个寄存器的值，根据其与零的比较结果判断是否跳转。该指令拥有 2MB 的寻址空间。需要注意的是，该指令不会影响处理器的条件码。

指令的汇编格式为“CBNZ <Xt/Wt>, <label>”。其中，Xt/Wt 表示用于比较的 64/32 位通用寄存器，label 表示该寄存器的值不为零时的跳转地址。

指令的编码如下：



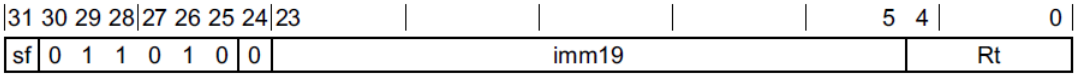

其中，sf 表示选用通用寄存器的 64 位（sf==1）或 32 位（sf==0），Rt 表示待比较寄存器，imm19 的作用与 B.cond 指令类似。因此，最终跳转地址的计算方式也是：<label> = pc + offset。

### 1.5.3 CBZ

CBZ 也是先比较，后跳转的指令，与 CBNZ 不同的唯一之处是跳转的条件不同。

指令的汇编格式为“CBZ <Xt/Wt>, <label>”。其中，Xt/Wt 表示用于比较的 64/32 位通用寄存器，label 表示该寄存器的值为零时的跳转地址。

指令的编码如下：



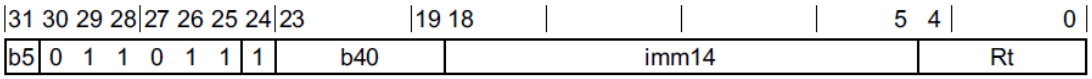

除了第 24 位外，其余指令与 CBNZ 完全相同，这里不再赘述。

### 1.5.4 TBNZ

TBNZ 是 ARM 指令集比较独特的指令，能够对寄存器中数值的某一比特位进行测试，根据测试结果判断是否跳转，拥有 64KB 的寻址空间。该指令不会影响处理器的条件码。该指令在实际可执行文件中较少出现，本次实验中可以不予考虑。

指令的汇编格式为“TBNZ <Xt/Wt>, #<imm>, <label>”。其中，Xt/Wt 表示待测试的 64/32 位通用寄存器，imm 表示需要测试的比特位编号，label 表示被测试比特位为 1 时的跳转地址。

指令的编码如下：



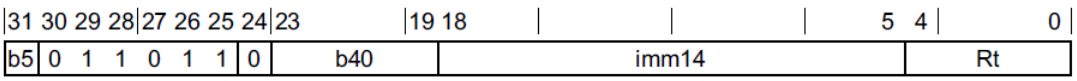

其中，b5 和 b40 拼接后表示待比较的比特位，即汇编格式中的#<imm>。imm14 的作用与 B.cond 指令类似，也需要左移两位并带符号扩展到 64 位作为 offset。因此，最终跳转地址的计算方式也是：<label> = pc + offset。

### 1.5.5 TBZ

TBZ 与 TBNZ 功能的唯一不同之处也在于分支条件的不同。

指令的汇编格式为“TBZ <Xt/Wt>, #<imm>, <label>”。其中，Xt/Wt 表示待测试的 64/32 位通用寄存器，imm 表示需要测试的比特位编号，label 表示被测试比特位为 0 时的跳转地址。

指令的编码如下：

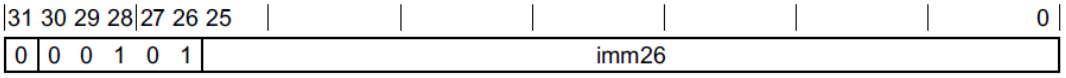



除了第 24 位之外，其余位的解释完全相同，这里不再赘述。

### 1.5.6 B

B 指令是直接分支、直接跳转的指令，拥有 256MB 的寻址空间。指令的汇编格式为“B <label>”，执行改指令会直接跳转到 label 的地址。

指令的编码如下：

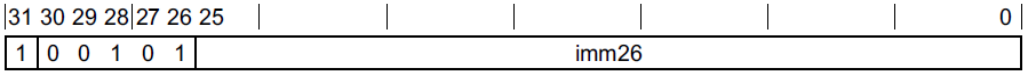



imm26 需要左移 2 位并带符号扩展到 64 位后作为偏移量 offset。label = pc + offset 仍然成立。

### 1.5.7 BL

BL 指令与 B 的唯一区别在于，在跳转的同时会把紧邻的下一条指令（PC+4）保存到 X30 寄存器中，通常与 RET 指令一起使用。BL 的汇编格式为“BL <label>”。

指令的编码如下：



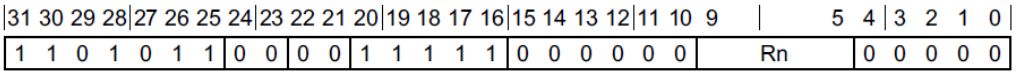

除了第 31 位外，其余编码与 B 指令完全相同。

### 1.5.8 BR

BR 指令采用寄存器寻址，因此拥有 64 位的跳转范围。当需要跳转的地址与当前 PC 的偏移量大于 B 指令的寻址范围时，比如跨用户态和内核态跳转时，才用该指令取代 B 指令。

BR 指令的汇编格式为“BR <Xn>”，Xn 中存储了跳转的目的地址。

指令的编码如下：

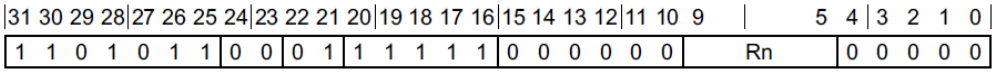



其中 Rn 与汇编格式的 Xn 编码一致。

### 1.5.9 BLR

BLR 与 BR 的关系，同 BL 和 B 的关系完全一致。通常在跨用户态和内核态调用函数时，使用该指令。该指令通常与 RET 指令一起使用。BLR 的汇编格式为“BLR <Xn>”。

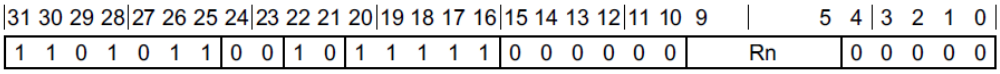
指令的编码如下：




### 1.5.10 RET

RET 指令与 BR 作用等价，写成 RET 是为了与 C 语言风格的 return 语句保持一致，便于阅读。指令的汇编格式是“RET {Rn}”，其中默认 Rn 为 R30，但也可以显式指定 Rn 为别的寄存器。

指令的编码如下：



### 1.5.11 ERET

ERET 指令是异常处理结束后返回用户进程的指令，需要借助 ELR 寄存器和 SPSR 寄存器寻址。本次实验不考虑异常处理过程，因此也不考虑 ERET 指令，感兴趣的同学可以进一步了解 ARM 处理器的异常处理过程。

指令的编码格式如下：

| 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 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 1 | 0 | 1 | 1 | | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | | 0 | 0 | 0 | 0 |

## 1.6 系统寄存器与系统寄存器指令

AArch64 架构存在系统程序员可见的上百个系统寄存器。每个处理器都有自己的一组系统寄存器。这些系统寄存器大都实现了权限隔离，也就是说，对于同一个系统寄存器，在不同的执行权限下实际会被映射到不同的系统寄存器中。比如，ARM 实现了 EL0，EL1，EL2 和 EL3 四个异常等级，把执行权限也分成相应四级，因此系统寄存器 TPIDR 就被实现成 TPIDR_EL0，TPIDR_EL1，TPIDR_EL2 和 TPIDR_EL3 四个系统寄存器。

ARM 提供了两个系统寄存器指令，用于处理器的系统寄存器与通用寄存器的数据传输，它们分别是 MRS 指令和 MSR 指令。需要注意的是，如果执行这两条指令时的权限低于相应系统寄存器的访问权限，比如在 EL0 下使用 MRS 读取 TTBR0_EL1 寄存器，就会产生非法指令异常。

### 1.6.1 MRS

MRS 指令用于把某个系统寄存器的值拿到指定的通用寄存器中，它的汇编格式为：“MRS <Xt>, <systemreg>”。由此可见，ARM 汇编的源地址在右，目的地址在左。

### 1.6.2 MSR

MRS 指令用于把指定通用寄存器的值写入某个系统寄存器中，它的汇编格式为：“MSR <systemreg>, <Xt>”。

## 1.7 系统寄存器与内存映射接口

除了通过 MRS 和 MSR 指令，ARM 还提供了从处理器外部访问系统寄存器的方法，即内存映射接口。通过内存映射接口访问的寄存器，大部分与调试相关，比如调试寄存器、CTI 寄存器、PMU 寄存器、ETM 寄存器等。

内存映射接口定义了上述寄存器的物理地址，该地址的计算方式为 pa = base + offset。其中，base 定义了一个寄存器集合（比如调试寄存器组、CTI 寄存器组等）的基地址，这个 base 是处理器厂商独立设计的，所以不同硬件设备的 base 值大都是不同的。offset 定义了具体寄存器在寄存器集合中的偏移，这个偏移量是由 ARM 官方规定的。因此，内存映射接口由 ARM 和处理器厂商共同决定。

在实际使用内存映射接口时，由于操作系统的存在，需要先把已知物理地址写入页表中，然后通过对应的虚拟地址访问。本次实验中，使用 Linux 的内核函数 ioremap 完成上述功能。

本次实验要求同学们找到树莓派 3B+的 CPU1-CPU3 的调试寄存器组物理基址。表 1-3 给出了 ARM 文档中规定的调试寄存器组的部分内存映射接口。

表 1-3 部分调试寄存器的内存映射接口

| Offset | Mnemonic | Offset | Menmonic |
|---|---|---|---|
| 0x020 | EDESR | 0xD00 | MIDR_EL1 |
| 0x080 | DBGDTRRX_EL0 | 0xD20 | ID_AA64PFR0_EL1 |
| 0x084 | EDITR | 0xD28 | ID_AA64DFR0_EL1 |
| 0x088 | EDSCR | 0xD30 | ID_AA64ISAR0_EL1 |
| 0x08C | DBGDTRTX_EL0 | 0xD38 | ID_AA64MMFR0_EL1 |
| 0x0400+16n | DBGBVR<n>_EL1 | 0xD40 | ID_AA64PFR1_EL1 |
| 0x0408+16n | DBGBCR<n>_EL1 | 0xD48 | ID_AA64DFR1_EL1 |
| 0xD50 | ID_AA64ISAR1_EL1 | 0xFA0 | DBGCLAIMSET_EL1 |
| 0xD58 | ID_AA64MMFR1_EL1 | 0xFB8 | DBGAUTHSTATUS_EL1 |

# 2 ARMv8 调试架构

ARM 的调试架构称为 CoreSight[7]。ARMv8 使用的 CoreSight 版本为 SoC-400。本次实验用到了其中的一小部分组件，比如调试寄存器和 CTI。下面对 ARMv8 的调试架构进行简单介绍。

## 2.1 调试方法

本实验中的调试对象为处理器，我们把它称为“调试目标”或“目标核”。按照调试过程是否影响了流水线的正常工作，调试可以分成入侵式调试和非入侵式调试。入侵式调试需要暂停并清空目标核的流水线，使目标核进入调试状态。非入侵式调试则在处理器正常执行的过程中收集数据，不对流水线产生任何影响。入侵式调试和非入侵式调试采用完全不同的方法实现，前者使用调试寄存器组完成，后者使用 PMU 和 ETM 等硬件单元完成。本次实验主要涉及包含硬件断点的入侵式调试。

ARMv8 处理器提供了三种调试机制，分别是外部调试、片上跨核调试和片上自调试。外部调试需要外接调试器，通过 JTAG 接口与片上的硬件单元交互，片上跨核调试使用同一芯片上不同的处理器对目标核进行调试，片上自调试是目标核自身通过异常（系统调用）进入高执行权限，然后由操作系统进行调试。本次实验主要使用了片上跨核调试机制。

## 2.2 调试状态

ARM 处理器的状态可以分成正常状态和调试状态。正常状态下，处理器通过流水线，以顺序发射、乱序执行、顺序提交的方式执行指令。而在调试状态下，处理器的流水线被清空，它的行为被调试器

完全接管。在片上跨核调试模式下，调试器是另一个处理器。调试器能够通过调试寄存器组控制处于调试状态的处理器，常见的行为包括单步执行指令和数据交互。

## 2.3 调试状态的进入和退出

ARM 提供了多种让处理器进入调试状态的方式，比如调试器通过 CTI 组件（一个与跨核信号生成与传输相关的硬件部件）发送异步的调试请求，或者调试器设置目标核的硬件断点，或者目标核自行执行 HLT 指令。本次实验主要使用了第二种方式，即硬件断点的设置。

调试寄存器中的 DBGBCR 和 DBGBVR 用于设置硬件断点。其中，DBGBCR 控制硬件断点的匹配方式，DBGBVR 保存了硬件断点的 64 位虚拟地址。由于硬件断点操作不当会导致机器死机，本次实验中的硬件断点设置已经实现好，同学们不需要修改。

调试器通过 CTI 给目标核发送重启请求，能够让处理器重启。具体的实现比较复杂，在 ARM 技术手册的 H5.6 章节中有所介绍。本次实验中的处理器重启也已经实现好，同学们不需要修改。

## 2.4 调试状态下单步执行指令

单步执行指令需要使用 EDITR 和 EDSCR 寄存器。它们都是 32 位的寄存器。调试器往 EDITR 寄存器写入一个 32 位的指令后，目标核将异步执行这条指令。EDSCR 寄存器中的 ITE 位和 ERR 位保存了这条指令的执行状态，ITE 为 0 说明指令写入 EDITR 后尚未被执行，为 1 说明指令已被执行完。ERR 则记录了指令是否成功执行。在调试状态下，访存类指令、异常触发指令、分支指令、数据运算指令等均不允许执行。这两个寄存器的物理地址见表 1-3。

## 2.5 调试状态下的数据传输

数据交互需要使用 DBGDTRTX 和 DBGDTRRX 两个寄存器，它们是 32 位寄存器，在物理空间上是连续的，可以拼接成一个 64 位寄存器 DBGDTR，一次性完成 64 位的数据传输。EDSCR 寄存器的 TXfull 记录了 DBGDTRTX 中的读写操作是否完成，RXfull 记录了 DBGDTRRX 中数据的读写操作是否完成，这两个标志的设置能够防止这两个寄存器内的值发生写覆盖，也就是后续的值把之前写入但尚未使用的值覆盖掉。表 2-1 表述了这两个标志的变化条件。表中，debugger 读写 DBGDTRRX 和 DBGDTRTX 的方式为内存映射接口，target 读写 DBGDTRRX 和 DBGDTRTX 的方式为 MRS 和 MSR 指令。

TXfull 防止了目标核写 DBGDTRTX 时的写覆盖。当目标核写 DBGDTRTX 时，TXfull 被置 1，说明 DBGDTRTX 中有目标核写的数据，且尚未被调试器接收。当调试器读 DBGDTRTX 时，TXfull 被清 0，说明目标核可以重新往 DBGDTRTX 写入数据。同理，RXfull 防止了调试器写 DBGDTRRX 时的写覆盖。这里不再赘述。在 32 位架构 AArch32 中，DBGDTRRX 和 DBGDTRTX 是单向工作的，前者只允许目标核写，调试器读，后者只允许调试器写，目标核读。因此才有了这种设计。在 64 位架构中，ARM 设计者保留了这种机制。由于 DBGDTRTX 和 DBGDTRRX 都变成双向通信，因此需要在使用时考虑二者的读写顺序。

表 2-1 TXfull 和 RXfull 标志的变化情况

| register | debugger read TRTX | debugger write TRRX | target read TRRX | target write TRTX |
|---|---|---|---|---|
| TXfull | clear to 0 | no change | no change | set to 1 |
| RXfull | no change | set to 1 | clear to 0 | no change |

调试状态下的数据传输有两种模式，分别是正常模式和访存模式。

在正常模式中，这两个寄存器是调试器与目标核寄存器的“中间人”，负责跨核传递数据。调试器通过这两个寄存器可以直接访问目标核的通用寄存器，并间接地读取目标核系统寄存器的值。图 2-1 简单展示了正常模式下数据传输的过程。

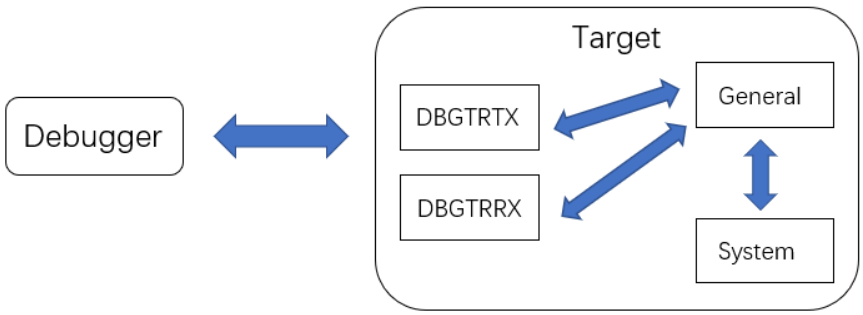



图 2-1 调试状态正常模式下数据传输

在访存模式中，这两个寄存器是调试器与目标核的内存空间的“中间人”，负责传递目标核的内存数据。调试器通过这两个寄存器读写目标核的内存空间。图 2-2 简单展示了访存模式下数据传输的过程。

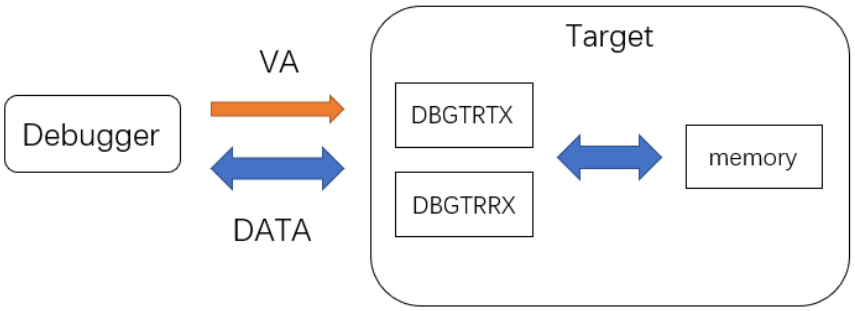



图 2-2 调试状态访存模式下数据传输

EDSCR 寄存器的 MA 位决定了当前调试器使用何种数据传输模式。

本次实验要求同学实现正常模式下的寄存器读写，以及访存模式下的内存读操作。ARM 技术手册里介绍了数据传输的流程[4]，正常模式读写的流程图在 H2.4 章节，内存读写的流程图在 K9.1 章节。这里把流程图给出。2.4 和 2.5 两节的内容足够用来解释流程图，这里不再给出更详细的说明。

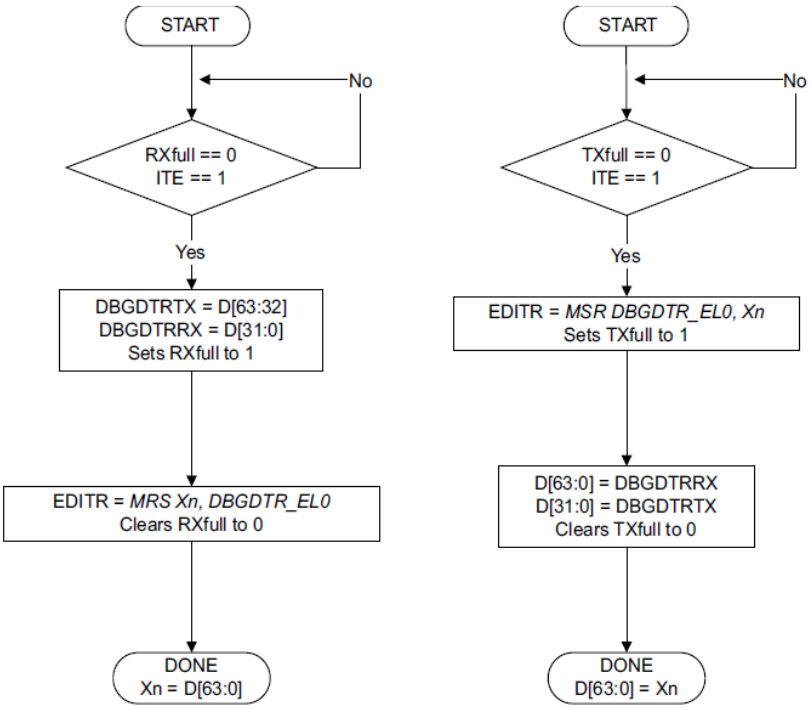



图 2-3 正常模式 64 位数据传输流程

（左边为 debugger->target，右边为 target->debugger）

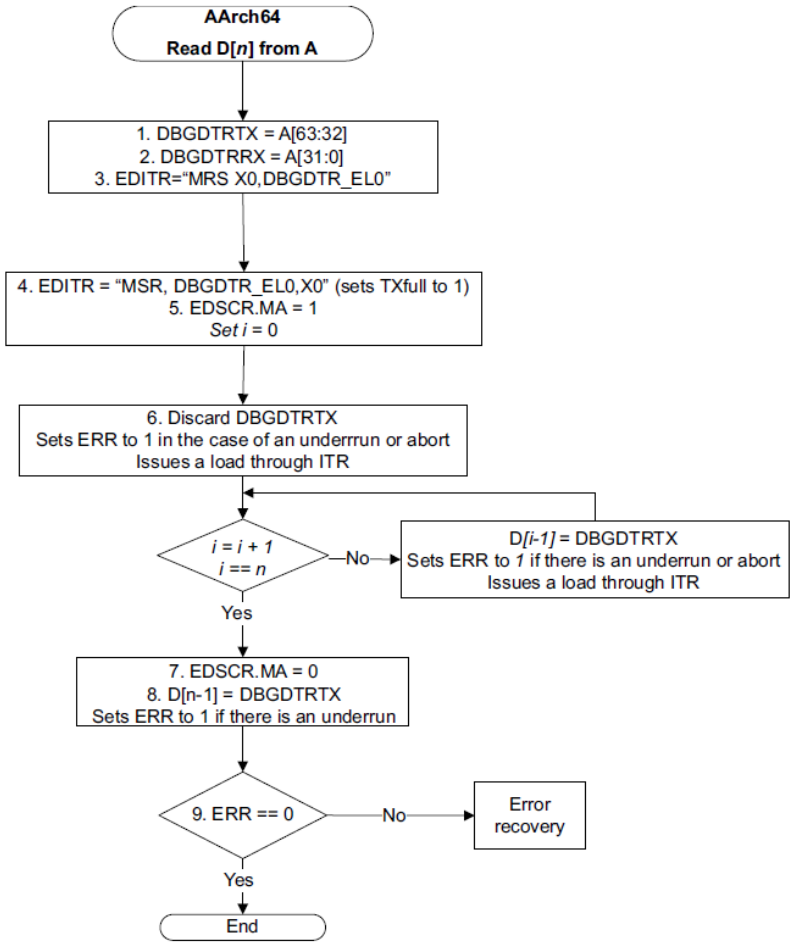



图 2-4 访存模式下从目标核的内存空间读 32 位数据的流程

## 2.6 调试架构的安全性（补充阅读）

ARMv8 调试架构存在很大的安全隐患。2019 年，韦恩州立大学的 Zhenyu Ning 和 Fengwei Zhang 提出一种利用 ARMv8 调试架构非法提升用户权限的攻击 Nailgun[5]。

为了实现权限隔离，ARMv8 实现了四个权限等级，在技术手册中称为四个“异常等级（Exception Level）”。其中，EL0 与 X86 的用户态类似，EL1 与 X86 的内核态类似，EL2 通常用于实现虚拟机的 Hypervisor，EL3 则用于实现 TrustZone 可信执行环境。Nailgun 攻击利用了片上跨核调试技术，通过操控一个处于 EL0 的处理器，使另一个处理器进入调试状态。在调试状态下，ARM 缺乏对处理器执行权限的基本检查，使得目标核的执行权限可以不受限制地提升到 EL3。攻击者可以借助 DCPS1-DCPS3 指令（这三条指令是调试状态下才允许执行的特殊系统指令）把目标核的执行权限提升到 EL1-EL3。之后，攻击者可以通过调试器和目标核之间的跨核数据传输，获取高特权内存空间下的私密数据。图 2-5 展现了攻击的主要流程。

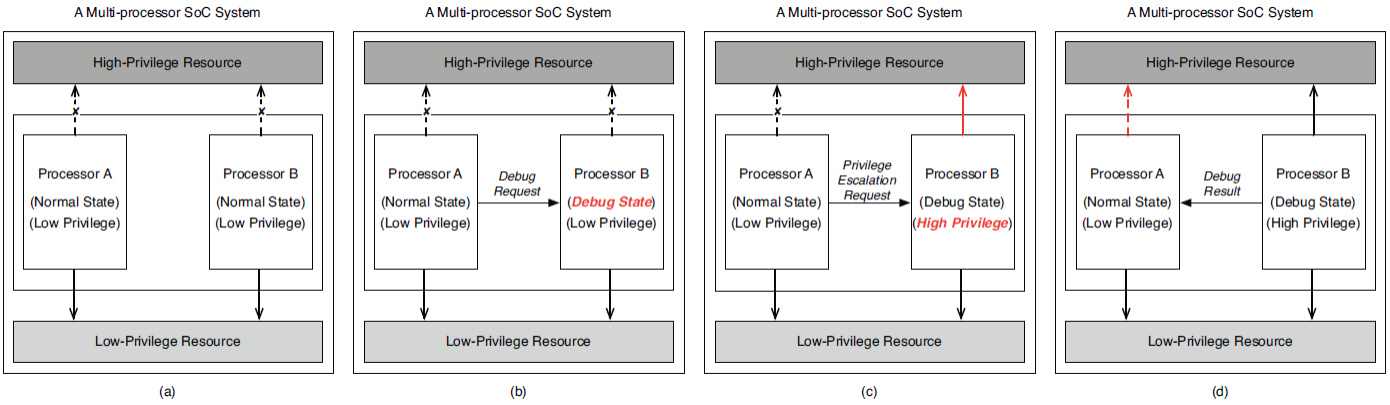



图 2-5 Nailgun 攻击

# 3 分支预测器的设计

为了解决控制流相关的问题，减小流水线阻塞造成的性能损失，现代处理器大都采用了分支预测技术。分支预测器通过记录分支指令的历史执行信息，对分支指令的跳转方向和跳转地址进行预测。分支预测器很好地提升了现代处理器的性能。

现代分支预测器的设计细节很少公开，有一些研究者通过逆向工程的工作还原了部分处理器的分支预测器设计。逆向工程不是本次实验需要考虑的内容，此外，本次实验不需要对分支预测器进行精准的模拟，同学们只需要依据在课程中所学的知识，实现简单的分支预测器即可。

## 3.1 分支预测器实例

本节介绍一种简单的分支预测器设计[6]，起到抛砖引玉的作用。图 3-1 介绍了这种分支预测器的组成。该分支预测器包括一个二级自适应分支预测部件和一个目标分支地址缓存区。全局历史记录寄存器（Global History Register，GHR）记录了分支指令的历史跳转信息。每个 GHR 的值可以和历史模式表（Pattern History Table，PHT）的一个条目相关联，PHT 根据 GHR 的值做出是否跳转的判断。BTB（Branch Target Buffer）记录了分支指令的地址和跳转地址的映射。如果 PHT 预测将跳转，则

从 BTB 中根据分支指令的地址查找对应的跳转地址。

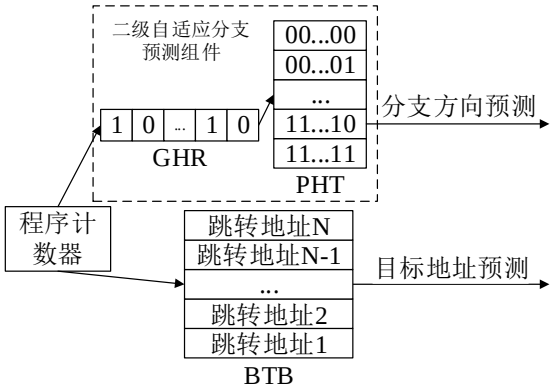



图 3-1 一个简单的分支预测器

此外，部分处理器，如 ARM，针对返回指令（如 ARM 的 RET 指令）的预测设计了特殊的 RSB（Return Stack Buffer），专门用于记录返回指令的跳转历史，从而对返回指令的跳转地址进行预测。RSB 的工作原理非常简单，在发生函数调用时，比如 ARM 处理器执行 BL 或 BLR 指令时，会把链接寄存器的地址压入 RSB 中。每次执行返回指令时，就会从 RSB 的顶部获取一个地址作为预测。ARM 处理器的 RSB 设计如图 3-2 所示。

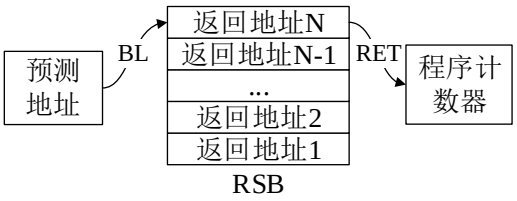



图 3-2 RSB 工作原理

## 3.2 分支预测漏洞（补充阅读）

现代分支预测器的正确率很高，但预测失败是不可避免的。当分支预测器预测失败时，为了不对程序执行的正确性带来影响，处理器不会提交预测错误的一系列指令。但是，这些被错误预测并执行的指令可能会在微体系结构状态中留下痕迹，比如把数据遗留在 Cache 中。攻击者可以利用这些痕迹恢复分支预测指令的执行结果。

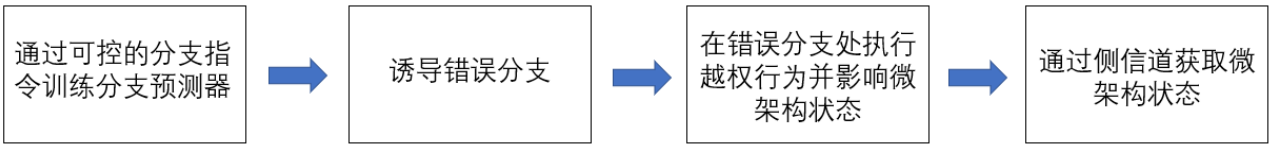



图 3-3 Spectre 攻击流程

Spectre 有许多变种，Spectre v1 的训练对象主要是 PHT，Spectre v2 的训练对象是 BTB，Spectre RSB 的训练对象是 RSB。

# 4 Linux 内核模块

ARM 片上调试需要内核权限，并且需要使用部分内核函数。本次实验通过添加内核模块的方式进入内核态，实现片上调试与指令跟踪。关于内核模块的开发，网上有大量的教程可供参考，并且本次实验中的内核模块注册、内核模块使用与内核模块编译的文件均已提供，因此这里只提供一些参考资料，不再进行细节上的介绍。

- 内核模块详解：https://blog.csdn.net/qq_33406883/article/details/100071183
- 设备驱动：https://www.cnblogs.com/downey-blog/p/10502235.html
- 内核线程：https://www.cnblogs.com/muahao/p/6210761.html
- 内核模块的命令行传参：https://blog.csdn.net/charcy/article/details/6830031

本次实验中，不要求同学们从零编写内核模块，但了解部分内核函数的使用有助于看懂实验代码。因此，这里简单介绍一些内核函数的使用。

## 4.1 内核的函数/线程绑定

在使用跨核调试的相关函数时，需要指定一个处理器作为调试器，在该处理器上绑定函数。否则，如果在目标核上调用这些函数，会导致内核发生段错误，甚至机器崩溃。借助内核函数 smp_call_function_single 可以很简单地实现函数绑定。该函数的声明如下：

```
void smp_call_function_single(unsigned int cpu, smp_call_func_t func, void* info, int wait)
```

其中，cpu 指定绑定的目标，0 表示绑定到 CPU0，依此类推。func 表示需要绑定的函数，info 表示需要传递给 func 的参数，wait 置为 1 即可。需要注意的是，func 函数的声明形式必须为：

```
type func(void* info)
```

否则无法通过编译。

在实验框架中还开启了一个监听线程，一旦监听到目标核进入调试状态，调试器就调用调试函数。因此，监听线程也要绑定到调试器所在的处理器上。内核线程的绑定只需要调用函数：

```
void kthread_bind(struct task_struct* k, unsigned int cpu)
```

其中，k 是内核线程的结构体指针，cpu 是绑定的目标。调用这个函数前需要对内核线程初始化，具体细节可以参考上面列出的相关资料。

## 4.2 IO 操作

Linux 提供了 io 操作，根据 io 设备的内存映射接口的虚拟地址，即可读写 io 设备或系统寄存器。本次实验用到了 ioread32，iowrite32，ioread64 和 iowrite64 四个函数。ioread32 函数的声明如下：

```
unsigned int ioread32(void __iomem* addr)
```

输入一个 io 设备的内存映射地址，返回对应地址的 32 位数据。ioread64 类似，只是返回值类型不同。

iowrite32 函数的声明如下：

```
void iowrite32(u32 value, volatile void __iomem* addr)
```

输入一个需要写入的 32 位数值和 io 设备的内存映射地址，即可完成写 io 设备操作。iowrite64 类似，只是 value 的类型不同。

### 4.3 设备驱动

为了实现用户态和内核态的数据交互，本次实验使用了设备驱动。除此之外，消息队列、Netlink 等机制也可用于用户态和内核态的数据交互，感兴趣的同学可以自行了解。

设备驱动利用了 Linux 文件系统的设计，这一块知识与本课程无关，感兴趣的同学可以在操作系统课上了解。在内核注册好设备之后，用户进程可以通过一般的读写操作访问内核空间的数据，具体细节可以参考上面列出的相关资料。在内核模块中，需要预先注册好设备操作的处理函数，并把它绑定到文件系统的某个文件索引节点下。比如，设备读操作的声明形式如下：

```
void device_read(struct file* flip, char* buffer, size_t len, loff_t* offset)
```

flip 为文件指针，buffer 是用户空间传入的缓冲区，len 是需要读取的字符长度，offset 是起始地址偏移。在用户进程调用 read 函数读设备文件时，就会根据文件索引调用内核态的这一函数，从而实现数据交互。从中可见，用户态和内核态通过设备驱动交互时，是以字节为单位传输数据。内核模块和用户进程都需要实现额外的格式化或类型转化函数解析数据。

### 4.4 打印调试信息

内核模块中使用 printk 函数输出调试信息。在命令行输入 dmesg 命令可以在内核日志中查看这些信息。dmesg 命令的常见用法：

- dmesg -c：打印内核日志后清空现有日志
- dmesg -wH：实时打印内核日志
- dmesg >> output.log：把内核日志追加写到文件 output.log 中

## 三 实验内容

本次实验分成两部分，第一部分是跨核调试与指令插桩的代码完善，另一部分是分支预测器的模拟和测试用例的构造。第一部分分成两个基本任务 task1 和 task2，第二部分分成两个基本任务 task3 和 task4。不同 task 之间的代码存在依赖关系，也就是说，需要先完成之前的 task，然后把结果填在之后的 task 中。进行 task1-task3 之前，请把代码中的 ID 换成自己的学号：

### 1 寻找树莓派 3B+调试寄存器组的基地址

task1 的代码在 task1 目录下。task1 的目录结构如下：

```
├── Makefile：用于编译内核模块
└── task1.c：内核模块代码
```

task1 的目标是找到树莓派 3B+的 CPU1，CPU2 和 CPU3 调试寄存器组的基地址。实验思路非常简单，只需要分别通过 MRS 指令和内存映射接口访问同样的一组寄存器，假设基地址是正确的，那么两次得到的值是匹配得上的。需要注意的问题是，通过 MRS 指令访问和内存映射接口访问时，需要用哪个处理器调用函数，以及这些函数获取的是哪个处理器的寄存器。在内核模块的入口 task1_init 中实现了上述思路，请同学们阅读代码，理解这两个函数各自的功能。task1 需要在 task1.c 的四处进行添加或修改。

第一处需要通过宏定义的方式定义五个 ID 寄存器的偏移，同学们可以在表 1-3 中找到部分 ID 寄存器的偏移，也可以在 ARM 技术手册中找别的寄存器。

```
/* TASK1 */
// TODO: choose another 4 ID registers, and find their offsets in documentation
#define MIDR_EL1                    0xD00
```

第二处需要修改记录 CPU 基地址的三个宏的值，它们分别是 DEBUG_REGISTER_ADDR_CPU1，DEBUG_REGISTER_ADDR_CPU2 和 DEBUG_REGISTER_ADDR_CPU3。目前已知 CPU0 的基地址（物理地址）为 0x40030000。

```
/* TASK1 */
// TODO: find debug register base address of CPU1, CPU2 and CPU3 [Note: default values below are not correct]
#define DEBUG_REGISTER_ADDR_CPU1    0x40000000
#define DEBUG_REGISTER_ADDR_CPU2    0x40000000
#define DEBUG_REGISTER_ADDR_CPU3    0x40000000
```

第三处在函数 access_register_via_system_instruction 中，需要通过系统寄存器访问指令 MRS 拿到预先定义的一些 ID 寄存器的值，通过系统指令拿到的值一定是正确的。拿到五个系统寄存器的值后，需要填到数组 val1 中。

```
static void access_register_via_system_instruction(void* junk) {
    reg64_t reg;

    /* TASK1 */
    // TODO: use system instrucion to access 5 ID registers, and store the value to val1[0]~val1[5]
    // hint: asm volatile(...); val1[X] = reg;

}
```

第四处在函数 access_register_via_memory_map 中，需要通过内存映射接口拿到预先定义的寄存器的值。需要注意的是，使用 ioread 函数访问错误地址有可能造成机器崩溃，因此在获取正确的基址前需要选择比较合理的可能地址。

```
static void access_register_via_memory_map(void* addr) {
    void __iomem* map_addr = (void __iomem*) addr;

    /* TASK1 */
    // TODO: use memory map to access 5 ID registers, and store the value to val2[0]~val2[5]
    // hint: val2[X] = ioread64(...);

}
```

下面给出一些提示。

- CPU0 的调试寄存器组的物理基地址是已知的，为 0x40030000
- 调试寄存器组的大小是已知的，为 0x1000

- SoC 设计中，调试寄存器通常在物理空间上是连续的。已知 CPU0-CPU4 的物理地址构成等差数列。
- 请在 0x40000000-0x50000000 的物理地址范围寻找。

预期的实验结果如下。在 task1 目录下执行 make，得到 task1.ko 文件，该文件是可执行的内核模块。在命令行执行命令“sudo insmod task1.ko”插入内核模块，然后执行“dmesg”查看内核日志，如果地址正确，那么应该输出以下结果：

```
module: check CPU0
CPU0: base address is correct
module: check CPU1
CPU1: base address is correct
module: check CPU2
CPU2: base address is correct
module: check CPU3
CPU3: base address is correct
```

假设存在地址错误，比如 CPU1 的基地址错误，那么会输出以下结果（具体数值未必一致）：

```
module: check CPU0
CPU0: base address is correct
module: check CPU1
CPU1: base address is not correct
debug: val1[0] = 0x410fd034, val1[1] = 0x2222, val1[2] = 0x10305106, val1[3] = 0x10000, val1[4] = 0x0
debug: val2[0] = 0x0, val2[1] = 0x0, val2[2] = 0x0, val2[3] = 0x0, val2[4] = 0x0
module: check CPU2
CPU2: base address is correct
module: check CPU3
CPU3: base address is correct
```

## 2 实现片上跨核调试中的数据传输

task2 的代码在 task2 目录下。task2 的目录结构如下：

```
├── Makefile：用于编译内核模块
├── debug_base.c：调试基础功能的实现，本次实验需要完善的代码在此
├── debug_base.h
├── debug_control.c：基于调试基础功能实现的函数，无需修改
├── debug_control.h
├── debugger.c：调试器的内核模块主程序，无需修改
├── register.h：记录了各个调试寄存器的位置，需要把 task1 的结果填到这个文件
└── testing：存放测试用例和执行脚本的文件夹，以下文件无需修改
    ├── empty：用户空间守护进程
    ├── run.sh：内核模块的执行脚本
    ├── test：task2 正确性测试用例的可执行文件
    └── test.c：task2 正确性测试用例的源代码
```

task2 的目标是实现跨核调试过程中的两种数据传输模式。进行 task2 之前，需要把 task1 中得

到的 CPU3 调试寄存器组基地址写到 register.h 中。

```
/* TASK1 */
// TODO: find debug register base address of CPU3.
// Note: You should pass task1 successfully and change the correct address. The machine will CRASH if you run the code with a wrong base address!
#define DEBUG_REGISTER_ADDR_CPU3
#define DEBUG_REGISTER_SIZE_CPU3            0x1000
```

task2 需要完成函数 read_64bit_from_target。该函数首先把某个通用寄存器中的数值写到 DBGDTR 寄存器（DBGDTRRX 寄存器和 DBGDTRTX 寄存器的拼接）中。接着检查 RXfull 或 TXfull，然后通过 ioread32 分别读这两个寄存器拿出数据。可以参考图 2-3 或代码注释完成函数。

task2 还需要完成函数 send_64bit_to_target。该函数首先通过 iowrite32 函数往 DBGDTRRX 和 DBGDTRTX 中写入数据，然后检查 RXfull 或 TXfull，最后把数据移动到目标核的某个通用寄存器中。可以参考图 2-3 或代码注释完成函数。

task2 还需要完成函数 read_memory_via_dcc。该函数访问目标核的某个虚拟地址处的 32 位数据。可以参考图 2-4 或代码注释完成函数。

预期的实验结果如下。完成代码编写后，进入 testing 目录，执行 run.sh。脚本会进行环境准备并插入内核模块，然后启动测试用例。如果一切顺利，命令行会输出如下结果：

```
ubuntu@ubuntu:~/lab3/break-point/answer2/testing$ ./run.sh
Congratulations, you have passed task2!
debugger exit
```

同时在内核日志会打印以下信息：

```
module init!
debug: test CoreSight
debug: CoreSight works
kthread1 start listening
debug: reg1 = 0xaa1a03e0, reg2 = 0x4869, reg3 = 0x2333
```

感兴趣的同学可以阅读用户态的测试用例 test.c，以及内核模块 debugger.c 的 listen 函数是如何进行测试的。测试的基本思路是在测试用例的某条指令打一个硬件断点，在这个断点插桩，并进行跨核数据传输。

如果实现错误，那么有可能出现：

- 仅输出 debugger exit，没有输出 Congratulations 信息，说明数据传输能工作，但输出的数据不正确。可以通过 dmesg 查看相应寄存器的值出了哪些错误。
- 机器崩溃，说明 ioread 和 iowrite 访问了错误地址，或者函数进入了某个死循环。

## 3 实现分支预测模拟器

task3 的代码在 task3 目录下。task3 的目录结构如下：

```
├── Makefile：用于编译内核模块
├── analyse
│   ├── debug
```

```
|   └── debug.c
├── branch_analyse.c：分支指令的解码和目标地址计算，无需修改
├── branch_analyse.h
├── branch_predictor.c：模拟分支预测器的实现，task3 需要完善的代码在此
├── branch_predictor.h
├── debug_base.c：调试基础功能的实现，需要把 task2 的代码填到这个文件
├── debug_base.h
├── debug_control.c：基于调试基础功能实现的函数，无需修改
├── debug_control.h
├── debug_device.c：用于生成分支指令 trace 文件
├── debug_device.h
├── debugger.c：调试器的内核模块主程序
├── tools.c：一些数学函数和格式转换函数
└── tools.h
├── elfreader：可执行文件解析工具，无需修改
|   ├── elf2struct.py
|   ├── elf2va.py
|   └── objparser.py：从可执行文件静态计算一些指令的地址
├── register.h：记录了各个调试寄存器的位置，需要把 task1 的结果填到这个文件
├── testing：测试用例，task4 需要往其中添加测试用例
|   ├── data.log：分支指令的 trace 文件，包含预测结果
|   ├── empty：用户空间守护进程
|   ├── empty.c
|   ├── run.sh：内核模块的执行脚本
|   ├── test1
|   ├── test1.c：测试用例 1
|   ├── … 其余测试用例
|   └── …
```

task3 的目标是模拟一个分支预测器，该分支预测器在给定一个分支指令的 PC 时，需要给出跳转的目的地址。本次实验限制分支预测器的大小为 1KiB。

task3 需要在 branch_predictor.c 中模拟分支预测器，不允许使用 malloc 等函数分配额外的内存空间，允许添加少量全局变量和局部变量。不允许引用外部文件的变量。

```c
void branch_predictor_init(void) {
    predictor_space = vmalloc(PREDICTOR_SIZE * sizeof(char));
    // TODO: other code if necessary
}

va_t get_from_branch_predictor(reg64_t pc) {
    // TODO: once your branch predictor get an instruction address, it should give a predicted target address
    return 0;
}

void branch_predictor_update(reg64_t pc, va_t target) {
    // TODO: the debugger will give the correct target address once a branch instrucion executed, and you need to update the branch predictor
}

void branch_predictor_release(void) {
    vfree(predictor_space);
    // TODO: other code if necessary
}
```

实现分支预测器后，进入 testing 目录，执行“./run.sh [testcase]”，其中 testcase 为 elf 格式的测试用例。那么，内核日志会输出分支预测的情况：

```
module init!
debug: test CoreSight
debug: CoreSight works
kthread1 start listening
pre_times = 147, direction_correct_times = 118, address_correct_times = 0
```

testing 中现有的测试用例只是为了给同学们调试代码，不作为最终正确率的评价指标。同学们可以自行设计并添加额外的测试用例。为了让分支指令可控，不建议使用除 stdio 等标准库外的其余库函数。现有工具也不跟踪内核空间的分支指令。

在最终的线下实验检查时，助教会提供统一的测试用例，作为分支预测正确率的评价标准。

为了方便同学们的调试，工具利用设备驱动实现了一个简单的通讯机制，把 trace 到的分支指令，正确的预测结果以及分支预测器的预测结果均打印到 testing 目录下的文件 data.log 中。输出规模限制为 50MB，超出规模的数据会覆盖原先的数据。同学们可以选择 printk 打印调试信息，也可以在 debugger.c 的 listen 函数中修改 98-104 行的数据格式，把自定义格式的数据写到设备驱动 /dev/test 中，然后修改 analyse 目录下的 debug.c，输出相应的数据。

## 4 设计测试用例

task4 也在 task3 目录下进行。同学们需要在 testing 文件夹中添加一个测试用例。该测试用例将被用于测试所有同学分支预测器的预测正确率。

# 四 评价标准

本次实验共 15 分。

## 1 评分细则

表 4-1 评分细则

| 任务 | 子项 | 评分 |
|---|---|---|
| task1 | 找到 CPU1-CPU3 的调试寄存器组基址 | 1 |
| task2 | 实现跨核调试中的寄存器读写 | 2 |
| | 实现跨核调试中的内存读取 | 2 |
| task3 | 设计一种分支预测器 | 2 |
| | 分支预测的正确率[a] | 3 |
| task4 | 设计测试用例 | 1 |
| report | 分支预测器的设计说明 | 1 |
| | 测试用例的说明 | 1 |
| | 回答问题[b] | 2 |

[a]见第 2 小节，[b]见第 3 小节

## 2 正确率评价

分支预测的正确率由分支方向正确率和分支地址正确率决定。对于单个测试用例，设分支方向预测正确率为$\alpha$，分支地址预测正确率为$\beta$，那么评分为 $min\{k_1\alpha+\beta,3\}$。$k_1$ 在综合考虑所有同学的实验结果后决定。

最终测试时，用所有同学在 task4 提供的测试用例测试。在统计正确率的平均值时，会删去使所有同学的平均正确率低的$k_2$个测试用例的测试结果。$k_2$在综合考虑所有同学的实验结果后决定。

## 3 回答问题

请从以下几个问题中选择三个感兴趣的问题回答。

（1）比较 ARM A64 指令集和你熟悉的一种指令集，说明两者各自的优势和不足。

（2）是否可以设计出一个不受 Spectre 攻击的分支预测器？如果可以，你认为至少需要增加多大的存储开销？

（3）你认为是否要依据分支指令的类型设计不同的分支预测部件？请解释原因。

（4）请阅读 Nailgun 攻击论文[5]的第 III 节，说明为何 ARM 调试架构是不安全的。

（5）在实现跨核数据传输的时候（task2），请分析 DBGDTRRX 和 DBGDTRTX 的不同读写顺序对 TXfull 和 RXfull 标志的影响，如何使用这两个标志防止 64 位数据传输时的写覆盖？

（6）Linux 内核模块与用户进程的数据交互方式有哪些？请介绍三种以上的方法。

（7）假设有一台 ARM 处理器的裸机（无操作系统），能在外部调试器的支持下，对该处理器的所有寄存器进行读写操作。你认为在裸机上进行本实验的 task1-task3 有无可能？请说明原因。

# 五 附录

本次实验需要配置树莓派。网上有较多树莓派 3B+的配置教程，这里只介绍大致的流程，具体细节请查看网上教程。树莓派配置的大致流程如下：

（1）下载 ubuntu20.04 镜像：https://ubuntu.com/download/raspberry-pi

（2）使用工具把镜像烧录到 SD 卡中，比如 windows10 下的 win32diskimager 工具。

（3）烧录后，在 SD 卡根目录创建一个空的 ssh 文件。

（4）插入 SD 卡，使用有线网连接树莓派，然后通过 ssh 访问树莓派，初始时树莓派的账户和密码都是 ubuntu。

（5）一些常规的配置，比如修改密码、修改并更新软件源等。

（6）建议把树莓派连入局域网，比如通过电脑或手机的热点与树莓派建立无线连接。配置无线连接的教程：https://blog.csdn.net/zhou865612640/article/details/106528675/

（7）使用 apt 下载 make，gcc。

通过以上步骤即可准备好实验环境。

# 六 参考资料


[1] https://www.raspberrypi.org/blog/why-raspberry-pi-isnt-vulnerable-to-spectre-or-meltdown/

[2] https://ubuntu.com/download/raspberry-pi

[3] https://developer.arm.com/documentation/dui0801/j/A64-General-Instructions/ADRP?lang=en

[4] https://developer.arm.com/documentation/ddi0487/ga

[5] Ning, Z., & Zhang, F. (2019, May). Understanding the security of arm debugging features. In 2019 IEEE Symposium on Security and Privacy (SP) (pp. 602-619). IEEE.

[6] http://kns.cnki.net/kcms/detail/61.1069.T.20210407.1425.002.html.

[7] https://developer.arm.com/documentation/100536/0302