中国科学技术大学

学士学位论文

SPARCv8 指令集

在 Coq 中的形式化建模

作者姓名: 王佳玮

学科专业: 计算机软件与理论

导师姓名: 付明 副研究员 冯新宇 教授

完成时间: 二〇一七年六月

University of Science and Technology of China

A dissertation for bachelor's degree

Formalizing SPARCv8 Instruction Set Architecture in Coq

Author's Name: Jiawei Wang

Speciality: Computer software and theory

Supervisor: A/Prof. Ming Fu Prof. Xinyu Feng

Finished Time: June, 2017

致　谢

感谢付明老师对论文的指导以及帮助。在这 4 个月的时间里，付明老师对我有过多次的指导，就形式化建模方面有过大量的讨论。是他带领我走进了形式化建模的大门，教会了我怎样去定义语法，操作语义，掌握基本的研究方法。使我在短短几个月的时间里，从没有任何相关背景知识到能够了解并完成整个建模工作。论文的材料收集、构思到最后的修改过程，都得到了付明老师的帮助，在此表示感谢。

感谢冯新宇老师提供了软件安全实验室这样一个环境。这里有热心帮助我解决各种问题的师兄，还有舒适的工作环境能够让我静下心来思考问题。在此表示衷心的感谢。

目　录

摘要 7

Abstract 9

第 1 章　绪论 10

1.1 研究动机 10

1.2 国内外相关工作 11

1.3 主要贡献 12

1.4 论文组织结构 13

第 2 章　相关技术背景 14

2.1 SPARCv8 简介 14

2.1.1 例：三个整数相加 15

2.1.2 寄存器 16

2.1.3 控制转移 18

2.1.4 窗口寄存器 19

2.1.5 陷阱 21

2.1.6 模式 23

2.1.7 写延迟 23

2.1.8 指令集 23

2.1.9 本节小结 24

2.2 证明工具 Coq 简介 25

2.2.1 归纳定义 25

2.2.2 定义非递归函数和递归函数 26

2.2.3 证明对象和证明脚本 · · · 26

第 3 章 SPARCv8 指令集的形式化建模 · · · 28

3.1 汇编指令语法 · · · 28

3.2 机器状态相关定义 · · · 30

3.2.1 寄存器文件 · · · 31

3.2.2 帧寄存器链表 · · · 33

3.2.3 延迟寄存器链表 · · · 35

3.2.4 机器状态相关定义 · · · 36

3.3 操作语义 · · · 37

3.3.1 单步执行和多步执行 · · · 38

3.3.2 中断，陷阱执行和模式选择 · · · 39

3.3.3 延迟状态转移和无效指令状态的转移 · · · 42

3.3.4 汇编指令的操作语义 · · · 44

3.3.5 指令异常处理 · · · 48

3.4 本章小结 · · · 51

第 4 章 形式化模型的性质及证明 · · · 53

4.1 确定性 · · · 53

4.2 隔离性 · · · 54

4.3 本章小结 · · · 55

第 5 章 验证窗口溢出处理函数 · · · 56

5.1 窗口溢出 · · · 56

5.2 窗口上溢的处理 · · · 57

5.3 窗口上溢处理的数学规范 ········ 60
5.3.1 前条件 ········ 60
5.3.2 后条件 ········ 62
5.4 正确性验证 ········ 62
5.5 本章小结 ········ 63
第 6 章 Coq 实现 ········ 64
6.1 汇编指令语法和记号 ········ 64
6.2 汇编指令的操作语义 ········ 66
6.3 模型性质相关定理 ········ 67
6.4 验证窗口溢出处理函数 ········ 68
6.5 本章小结 ········ 70
第 7 章 总结 ········ 71
7.1 本文工作总结 ········ 71
7.2 进一步工作 ········ 71
参考文献 ········ 73

图目录

2.1 PSR 寄存器和 TBR 寄存器各段名称 · 17
2.2 3 个互相重叠的窗口和 8 个全局寄存器 · 21
2.3 窗口数为 8 时的窗口寄存器 · 22
3.1 SPARCv8 汇编指令的抽象语法 · 28
3.2 寄存器文件的定义 · 31
3.3 帧寄存器链表的定义 · 33
3.4 例：窗口左旋 1 次 · 34
3.5 延迟寄存器链表的定义 · 36
3.6 机器状态的定义 · 36
3.7 代码堆和事件的定义 · 37
3.8 操作语义建模的结构 · 37
3.9 多步执行 · 38
3.10 中断，执行陷阱和模式选择 · 41
3.11 延迟状态和无效指令状态的转移 · 43
3.12 简单汇编指令的操作语义 · 44
3.13 简单汇编指令的操作语义（续） · 45
3.14 复杂汇编指令的操作语义和异常处理 · 52
5.1 将窗口寄存器划分为栈空间和当前窗口 · 56
5.2 保存和恢复上下文时栈的变化 · 57
5.3 处理窗口上溢 · 58

5.4 窗口上溢陷阱处理过程 · 62

6.1 Coq 代码结构和代码量统计 · 64

表目录

2.1 例：含有延迟控制转移指令对的代码片段 · 19
2.2 例：执行延迟控制转移指令对时所产生的控制转移情况 · · · · · · · · · · · · 20
2.3 寄存器地址映射 · 20
3.1 通用寄存器及对应别名 · 29

摘 要

嵌入式操作系统被广泛应用于航空航天等安全攸关领域，它们通常采用 C 内嵌汇编的方式实现。用形式化验证技术证明 «««< HEAD 嵌入式操作系统实现的正确性，不可避免地需要考虑对不同硬件平台下的汇编指令集的行为进行形式化建模。SPARCv8 是目前嵌入式控制系统常用的处理器标准版本, 在安全攸关的航天设备的系统中得到广泛应用。

中国航天科技集团公司五院 502 所基于 SPARC 处理器自主研发的嵌入式操作系统 SpaceOS2 在航天任务中被广泛使用。对 SpaceOS2 进行形式化验证并提供可信编译的支持，从而提高 SpaceOS2 的安全性和可靠性，需要我们对 SPARCv8 指令集进行形式化建模。本文结合 SPARCv8 的指令手册，给出了 SPARCv8 指令集的抽象语法定义，抽象的机器状态，和指令集在机器状态上对应的操作语义；并证明了操作语义上的确定性和隔离性等性质。此外，我们基于操作语义验证了 SPARCv8 体系架构下特有的窗口上溢处理函数对应的 31 行汇编代码的行为满足用户的期望。

本文所有的形式化建模和证明工作都在定理证明工具 Coq 中进行了实现。

关键词: SPARCv8 Coq 形式化建模 操作语义 隔离性 窗口上溢

======= 嵌入式操作系统的实现的正确性，不可避免的需要考虑对不同硬件平台下的汇编指令集的行为进行形式化建模。

SPARCv8 是目前嵌入式控制系统常用的处理器标准版本, 在安全攸关的航天设备的电子系统中得到广泛应用。中国航天科技集团公司五院 502 基于 SPARC 处理器自主研发的嵌入式操作系统 SpaceOS2 在航天任务中被广泛使用，为了对 SpaceOS2 进行形式化验证以及提供可信编译的支持，从而提高 SpaceOS2 的安全性和可靠性，需要对 SPARCv8 指令集进行形式化建模。本文结合 SPARCv8 的指令手册，给出了 SPARCv8 指令集的抽象语法定义，抽象的机器状态，和指令集在机器状态上对应的操作语义；并证明了操作语义上的确定性和隔离性等性质。此外，我们基于操作语义验证了 SPARCv8 体系架构下特有的窗口上溢处理函数对应的（30 行?）汇编代码的行为满足用户的期望。本文所有的形式化建模和证明工作都在定理证明工具 Coq 中进行了实现。

关键词: SPARCv8 Coq 形式化建模 操作语义 隔离性 窗口上溢 »»»>

5953707f69b5db5989e5b20d6a5c14d9ac32857a

Abstract

Embedded operating systems are widely used in aerospace and other security related areas. They are usually implemented in C inline assembly. In order to use formal verification technology to prove the correctness of embedded operating system, it is inevitable to consider formalizing the behavior of the assembly instruction set under different hardware platforms. SPARCv8 is a standard version of the processor currently used in embedded control systems, and it is widely used in security-related aerospace equipment systems.

Based on the SPARC processor, China Aerospace Science and Technology Corporation (CASC) developed an embedded operating system called SpaceOS2. It is widely used in space missions. In order to formal verificate SpaceOS2 and provide support for trusted compilation, we need to formalize the SPARCv8 instruction set. In this paper, we give the abstract syntax definition of SPARCv8 instruction set, the abstract machine state, the operation semantics related to the machine state. We also prove the deterministic and isolation property with respect to the operational semantics. In addition, we verify 31-line assembly code, a window overflow handler function in SPARCv8 satisfies the user's expectations. The assembly code is a window overflow handler function in SPARCv8.

All of the formal modeling and proof work in this paper are implemented in the proof assistant tool Coq.

Key Words: SPARCv8, Coq, Formalization, Operational Semantics, Isolation, Window Overflow

第 1 章　绪论

本章首先介绍本文工作的研究动机，然后与国内外的相关工作进行比较，接着给出了本文工作的主要贡献，最后列出了本文的组织结构。

1.1　研究动机

随着计算机系统在国防、金融等领域中得到广泛使用，构建高可信系统在计算机系统的发展中扮演着越来越重要的角色。操作系统内核作为操作系统的核心部分，其可靠性是构建高可信计算机系统的关键。防崩溃代码 (Crash-Proof Code) [2]，即用形式化验证技术严格保证底层操作系统的正确性，被 2011 年 MIT 出版的《技术评论》评选为十大新兴技术之一，成为了一个新的研究热点。

在航空航天等安全攸关领域，底层的操作系统通常由 C 语言和内嵌汇编来实现。已有操作系统验证项目 CertiμC/OS-II [32] 和 seL4 [18, 25] 等为了简化对抽象机器的形式化建模，通常不对汇编指令集合进行建模，而是在抽象状态上刻画汇编代码的行为，从而避免了在 C 程序状态上暴露底层的机器细节，如寄存器，栈等。这也导致内核代码中的汇编代码实际上并没有被验证，而只是在抽象状态给了它一个规范，如果要验证这部分汇编代码满足这个抽象规范，不可避免地需要对汇编指令的具体行为进行形式化建模。

本论文工作选择对 SPARCv8 [6] 指令集进行形式化建模，是源于实时嵌入式内核 SpaceOS2 [5] 验证项目的需求。SPARCv8 作为一款高性能、高可靠、低功耗的微处理器，在航空航天，通信以及各种各样的嵌入式应用领域方面被广泛应用。SpaceOS2 由中国航天科技集团公司五院 502 所研发，被部署在嫦娥三号探月任务控制计算机等设备上面，它其中的汇编代码采用 SPARCv8 指令集实现，要完成这部分汇编代码的验证，需要首先对 SPARCv8 指令集进行形式化建模，给汇编代码一个数学语义模型。另外，实验室之前的操作系统内核验证工作 [32] 主要集中在 C 代码层面上，为了能保证目标汇编代码和 C 源代码之间行为的一致性，希望通过使用经过验证的可信编译器 CompCert [27, 28] 来编译保证。然而 CompCert 目前只支持将 C 的一个重要子集 Clight 翻译为 PowerPC [4]、ARM[1] 和 x86[11] 汇编指令集，并不支持 SPARCv8。为了能扩展 CompCert 使其后端能够支持 SPARCv8，需要对 SPARCv8 指令集进行形式化建模。

1.2 国内外相关工作

目前国内外对汇编指令集的形式化建模工作主要针对的是目前比较流行的指令集 x86 和 ARM，而对 SPARC 指令集进行形式化建模的工作比较少。与 ARM 和 x86 的形式化建模相比，对 SPARCv8 指令集的建模存在着更多的挑战，这归根于 SPARCv8 架构中的一些不同的特性。如在 ARM 和 x86 中分别有 16 个和 14 个通用整数寄存器，但在 SPARCv8 中，窗口数量为 8 时，就有 8×16 个窗口寄存器和 8 个全局寄存器作为通用整数寄存器。并且在运行过程中，每次只能接触到其中的 24 个窗口寄存器。如何更清晰的为这一特性建模，将其分为不同的层次，对接触不到的寄存器进行隐藏，也是建模中的一个挑战。除此之外，SPARCv8 还有一些其他特性，如使得函数跳转更为灵活的延迟跳转机制，可将应用程序代码与操作系统代码在物理层次分开的模式机制，以及模式切换方式——陷阱等。怎样清晰的展示这些特性也是在形式化建模中需要考虑的问题

英国剑桥大学的 Fox 和 Myreen 等人给出了 ARMv7 汇编指令的模型 [22]，使用了单子规范 (Monadic Specifications) 对指令解码，操作语义等性质进行了建模。瑞典皇家理工学院的 Narges Khakpour 等人在 HOL4 定理证明工具中证明了 ARMv7 中与安全相关的性质，包括内核安全，用户模式隔离等。微软研究院的 Andrew Kennedy 等人在 Coq [9] 中对 x86 的子集进行了形式化建模 [24]，使用了 Coq 中类型类（Type Classes）、记号（Notation）等特性，并使用了 Coq 中的数学库——Ssreflect [7]。除此之外，在 CompCert 编译器中，同样存在对 ARM 和 x86 的形式化建模 [8]。由于之前介绍中提到的 SPARCv8 的众多特性，上述这些关于 x86 和 ARM 指令集的建模工作不能直接应用在 SPARC 指令集的形式化建模中。

另外，与本为工作较为接近的是南洋理工大学的 Zhe Hou 等人在 Isabelle 中对 SPARCv8 指令集进行了形式化建模 [23]，但其目的是为了把经过验证的形式化模型转为可执行代码从而来研究 SPARCv8 处理器，不能用来证明汇编代码的正确性、也无法用来扩展可信编译器 Compcert 的后端。证明汇编代码要求我们对其中的语法和操作语义的规则有更佳清晰的定义，机器状态定义需要有一定的层次性，从而方便对汇编代码的证明。除此之外，他们没有对 SPARCv8 中的中断进行建模，不能描述操作语义由于中断到来而造成的不确定性。我们对 SPARCv8 指令集合的建模在定理证明工具 Coq 中完成，而 Compcert 也是使用 Coq 实现的，便于实现对 CompCert 后端的扩展。此外，我们还基于该形式化模型对窗口上溢处理函数进行了验证。

一些在对 x86 汇编代码进行形式化验证的工作 [19–21] 对 x86 简化指令集进行建模，也考虑对中断管理进行形式化建模，这些建模工作主要是为了研究 x86 汇编代码的验证技术，考虑的指令集比较小，模型相对比较简单。而本文的形式化建模工作对 SPARCv8 指令集的特性考虑相对比较完整，包括 SPARCv8 中窗口、中断和陷阱机制的形式化建模。

在建模及 Coq 实现过程中，除了参考上述工作外，还参考了 Krebbers [26]、Atkey [13]、Czarnik [17]、Benzaken [14]、Chlipala [16]、Xiaomu Shi [30]、Affeldt[12] 等人在 Coq 中对 C 指令集，Java 指令集等进行形式化建模的工作。

1.3 主要贡献

本文首先对 SPARCv8 指令集进行形式化建模，而后证明了与模型相关的一些性质，从而在一定程度上检验了模型的正确性。最后使用此形式化模型证明了窗口溢出处理函数的代码片段，从而说明了此模型的可用性。具体贡献如下：

1. 对 SPARCv8 指令集的行为进行形式化建模，建模忠实于 SPARCv8 用户手册 [10] 中对各条指令的行为描述，囊括了 SPARCv8 相比与其它指令集（如 x86, ARM）而特有的行为特性，包括提高函数调用效率的窗口寄存器机制，使得函数跳转更为灵活的延迟跳转机制，可将应用程序代码与操作系统代码在物理层次分开的模式机制，以及特殊的模式切换方式——陷阱。

2. 基于上述形式化建模提供的操作语义，我们证明了该机器模型的语义行为在没有外部中断的情况下满足确定性，还证明了用户模式和管理者模式下的代码执行满足隔离性。

3. 本文以窗口溢出处理函数的 31 行代码片段为例，给出了代码片段的形式化规范（前后条件），基于操作语义证明了该代码片段满足给定前后条件，并且执行中不产生任何异常。

4. 本文描述的所有工作都在定理证明工具 Coq 中进行了实现。其中关于 SPARCv8 的形式化建模定义 947 行，确定性和隔离性的证明 1611 行，验证窗口溢出处理函数 7114 行，总计 9672 行。源代码可以通过如下链接访问：`https://github.com/wangjwchn/sparcv8-coq`。

1.4 论文组织结构

- 第二章介绍了 SPARCv8 指令集的主要特性和形式化建模相关的技术背景。
- 第三章给出了 SPARCv8 指令集合的形式化建模，包括指令集的语法、机器状态和操作语义。
- 第四章给予第三章的形式化模型，证明了 SPARCv8 语义的确定性和隔离性两个重要性质。
- 第五章基于 SPARCv8 的形式化模型验证窗口溢出处理函数。
- 第六章简要介绍了论文工作对应的 Coq 实现。
- 第七章对论文工作进行了总结，并列出了基于该工作未来将开展的工作。

第 2 章　相关技术背景

本章将简要介绍 SPARCv8 处理器指令集的主要特性，以及对 SPARCv8 进行形式化建模的定理证明工具 Coq。SPARCv8 相关的介绍在第一节中给出，Coq 相关的介绍在第二节中给出。

2.1　SPARCv8 简介

SPARC，即可扩充处理器架构（Scalable Processor ARChitecture），是国际上流行的 RISC 处理器体系架构之一，最早于 1985 年由 Sun 公司所设计，是业界出现的第一款具有可扩展性功能的微处理器。SPARC 既可以应用于高性能工作站和服务器，又可以应用于嵌入式设备。SPARC v7/v8 是目前嵌入式控制系统常用的处理器标准版本，在航天设备的电子系统中得到广泛应用。例如，欧洲空间研究与技术中心研发的基于 SPARCv8 架构的处理器——LEON3 [3]，广泛应用在其航天局的各种专用集成电路芯片内。

与 x86, ARM 相比，SPARCv8 性能更好，功耗更低，更可靠。这得益于 SPARCv8 中一些独特的机制。如：

- 为了更灵活的函数跳转，而引入的多种控制转移指令和无效指令状态。
- 为了提高函数调用时上下文保存和恢复的效率，而引入的窗口寄存器和窗口旋转机制。
- 模式之间的切换方式——陷阱，既可用于应用程序主动调用内核函数，又可以用陷阱做一些错误的处理，比如除零错误，地址对齐错误等等，还可以用来处理中断请求。
- 为了将运行在处理器上的操作系统中的应用程序代码与内核代码在物理层次分开，而引入的模式机制。
- 写状态寄存器时，与实现有关的延迟写特性。
- 为了支持以上特性而引入的丰富的汇编指令。

本章先从这几个方面介绍 SPARCv8 指令集的特性。这些特性为形式化建模带来了一定的挑战。

2.1.1 例：三个整数相加

首先我们给出一个简单的例子——三个整数相加，来看一下 SPARCv8 中的部分特性。

代码 2.1 加法函数及调用者的 SPARC 汇编代码

```
1  ! 调用者:
2  !      int i;                    /* 编译器将l7寄存器分配给"i" */
3  !      i = sum3(1,2,3);
4              ...
5              mov   1, %o0         ! 传入sum3函数的第一个参数为1
6              mov   2, %o1         ! 传入sum3函数的第二个参数为2
7              call  sum3           ! 调用sum3函数
8              mov   3, %o2         ! 最后一个参数放入延迟槽中
9              mov   %o0, %l7       ! 将返回值拷贝到 %l7 中
10                                  ! (即变量"i")
11             ...
12
13 ! 被调用函数:
14 !      int sum3(a,b,c)           /* 传入参数放入 %i0,
15 !                                  %i1, 和 %i2 中 */
16 !             int a,b,c;
17 !      {
18 !             return a+b+c;
19 !      }
20
21
22 ! sum3 函数的第一种实现方式
23 sum3:
24        save  %sp,-64,%sp         ! 分配栈空间
25        add   %i0, %i1, %l7       ! 相加结果放入l7中
26        add   %l7, %i2, %l7       ! (也可直接使用i0)
27        ret                       ! 从sum3函数返回，并且
28        restore %l7, 0, %o0       ! 将结果放入o0中，恢复上下文
29
30
31
32 ! sum3 函数的第二种实现方式
33 sum3:
34        add   %o0, %o1, %o0
35        retl                      ! 从叶结点返回
36        add   %o0, %o2, %o0
```

代码 2.1给出了三个整数相加函数（sum3）的 SPARCv8 汇编语言的实现。函数调用者将三个参数分别放入 o0、o1、o2 寄存器中，调用 sum3 函数。sum3 函数返回时，将返回值放入 o0 中。代码中给出了两个 sum3 函数的实现，分别在第 22 ～ 28 行和第 32 ～ 36 行。

注意到，调用者在调用 sum3 时，先将前两个参数放入对应位置，而后调用 sum3 函数，最后再放入第三个参数，如第 5 ～ 8 行所示。原因是，SPARC 中的函

数调用 call 函数执行时，只会记录下将要跳转的地址，在下一个指令周期仍执行放在下一地址的一条指令（add %l7, %i2, %l7)，这条指令执行完之后，才真正执行跳转指令。也就是说，虽然 add %l7, %i2, %l7 指令在跳转指令 call sum3 的下方，但却在跳转前执行。

调用 sum3 函数后，sum3 函数可以选择是否保存父函数的上下文。如果需要保存上下文，则如第一种实现方式（第 22 ～ 28 行）所示。首先执行 save 指令。在 save 指令中，可以选择需要的栈空间大小（本例中为 64）。执行保存上下文指令后，原来在父函数中的 o0、o1、o2 寄存器的值为子函数的 i0、i1、i2 的值。此时将 3 者相加保存到 l7 中。接下来执行 ret 指令跳转到父函数，注意此处仍然不是直接跳转，而是延迟一条指令执行。所以最后会执行 restore 函数。restore 指令首先将 l7 中的值取出，而后恢复上下文，最后将 l7 的值写入父函数的 o0 中。

如果不保存上下文，则如第二种实现方式（第 32 ～ 36 行）所示。由于没有保存上下文，子函数仍有访问父函数中变量寄存器的权限，此时需要注意不能破坏父函数寄存器的值，故讲计算在 o0 寄存器中进行。注意此处的 retl 仍然是延迟执行的。

整数相加函数 sum3，短短几行代码就有很多需要注意的地方。SPARCv8 中的众多特性使得对编程人员的要求较高，需要编程人员对 SPARCv8 拥有很深的理解并拥有很强的编程能力，需要对汇编代码进行仔细查验，才能保证不出现错误。例如，在实际编程中为了可读性，将 sum3 第二种实现方式中的两条加法指令放在一起：

代码 2.2　为了可读性而改写的 sum3 函数

```
1  ! 为了可读性而改写的sum3函数
2  sum3:
3          add   %o0, %o1, %o0
4          add   %o0, %o2, %o0
5          retl                          ! 从叶结点返回
6          nop                           ! （十分重要！）
```

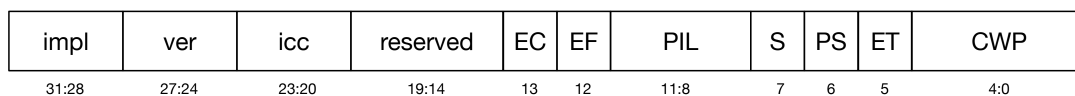
则在跳转指令（retl）之后，一定要接一条 nop 指令，否则延迟跳转会让跳转指令后面相临的一条不确定的指令执行，发生严重错误。

所以，对 SPARCv8 汇编指令进行形式化建模，并使用此模型验证编程人员所写代码的正确性，是十分必要的。

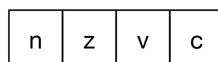
2.1.2　寄存器

SPARCv8 中主要有以下寄存器：

PSR



ICC



TBR

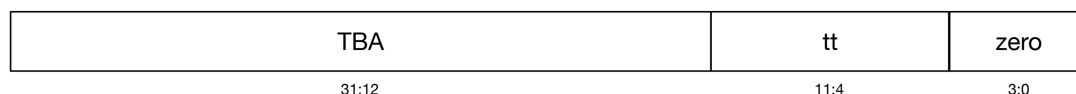


图 2.1　PSR 寄存器和 TBR 寄存器各段名称

- **PSR 寄存器**

PSR 寄存器为处理器状态寄存器，分为多个段，如图 2.1所示。PSR 寄存器包括与实现有关的 impl 段、与版本有关的 ver 段、用来描述计算状态的 n、z、v、c 段，分别用来表示上一次的计算结果是否为负数、是否为零、是否溢出、是否进位。还有保留的 reserved 段、表示是否开启协处理器的 EC 段、表示是否开启浮点寄存器的 EF 段、表示允许中断的等级的 PIL 段、表示当前模式的 S 段、表示之前模式的 PS 段、表示是否允许陷阱发生的 ET 段、表示当前窗口指针寄存器 CWP 段。

- **TBR 寄存器**

TBR 寄存器为与陷阱有关的寄存器，分为多个段，如图 2.1所示。TBR 寄存器包括表示陷阱表位置的 TBA 段、表示陷阱类型的 tt 段、值恒为 0 的零段。

- **全局寄存器和窗口寄存器**

全局寄存器共 8 个。窗口寄存器的数量与实现有关，在任何时刻只能操作其中的 24 个，其余的部分被隐藏，不可操作。

- **WIM 寄存器**

WIM 寄存器用来表示哪些窗口是可用的。如果 WIM 寄存器中第 n 位值为 1，表示标号为 n 的窗口不可用；值为 0 则表示可用。

- **Y 寄存器**

  与除法类指令相关的寄存器。

- **PC 寄存器和 nPC 寄存器**

  与控制转移相关的寄存器。

- **ASR 寄存器**

  辅助状态寄存器，用来临时保存 PSR 的值。

在上述定义的寄存器中，可以根据与用户交互的权限分为 3 个种类:

1. 用户不可直接操作的寄存器，如 PC、nPC 寄存器等等。用户没有指令可以对这些寄存器的值进行直接修改，但可以进行间接修改。如通过控制转移指令来间接修改 PC、nPC 寄存器的值。

2. 用户可操作的状态寄存器，指用户可以通过状态寄存器读写指令操作的寄存器，如 WIM、Y、ASR 寄存器等等。

3. 剩余的寄存器可被称为用户可操作的通用寄存器，即表示执行除状态寄存器读写指令外的指令时，可以控制的寄存器。通用寄存器通常在执行算数指令时使用，共 32 个。

2.1.3 控制转移

和 x86 架构或 ARM 架构不同，SPARCv8 中与程序的控制转移相关的寄存器有 2 个。分别为程序计数器（PC），下一个程序计数器（nPC）。同时还有无效指令状态（annuled）的特性。SPARCv8 中，控制转移类指令可分为 4 种，分别为无转移指令、条件延迟转移指令、延迟转移指令、非延迟转移指令。这 4 种控制转移类指令执行后，控制转移寄存器 PC 和 nPC 的值的变化分别为:

- 对于无转移指令，nPC 值赋值给 PC，nPC 自增 4。

- 对于延迟转移指令，nPC 值赋值给 PC，跳转地址赋值给 nPC。

- 对于条件转移指令，如果判定条件成立，则相当于一条延迟转移指令；如果判定失败，则相当于一条无转移指令。

- 对于非延迟转移，执行跳转地址赋值给 PC，nPC 为跳转地址加 4。

对于条件延迟转移，延迟转移两类指令还可能产生无效指令状态。这两类指令执行后，如果判定条件成立，则进入无效指令状态。进入无效指令状态的机器在一定条件下，下一条指令会直接跳过，同时系统恢复普通状态。

综上，对控制转移的建模需要三个寄存器，分别代表 PC、nPC 的值，以及是否处于无效指令状态。这三个寄存器的值便能刻画复杂的机器状态，如 2 条控制转移指令连续执行时的机器状态。考虑一个代码片段，其中有且仅有 2 条连续放置的控制转移指令（也称之为控制转移指令对），如表 2.1所示。则对于不同的控制转移指令，可能会有不同的转移过程，如表 2.2所示。

表 2.1　例：含有延迟控制转移指令对的代码片段

地址	指令	目标地址
8	非控制转移指令	
12	控制转移指令	40
16	控制转移指令	60
20	非控制转移指令	
24	...	
...	...	
40	非控制转移指令	
44	...	
...	...	
60	非控制转移指令	
64	...	

2.1.4　窗口寄存器

SPARCv8 中有若干个窗口，具体数量与实现相关。在任意时刻，一条指令能够访问到的通用寄存器的数量为 32 个（记为 r_0 至 r_{31}），分别由 8 个全局寄存器 (global) 和当前窗口所对应的 24 个窗口寄存器组成。当前窗口由当前窗口指针 (CWP) 决定。这 24 个窗口寄存器被分为 3 组，每组 8 个寄存器，分别为出寄存器组（out）、本地寄存器组（local）、进寄存器组（in），对应关系如表 2.3 所示。

表 2.2　例：执行延迟控制转移指令对时所产生的控制转移情况

情况	12: 控制转移指令 40	16: 控制转移指令 60	控制转移顺序
1	延迟转移	无转移	12,16,40,60,64,...
2	延迟转移	延迟转移判定失败	12,16,40,44,...
3	延迟转移	无转移，无效状态	12,16,44,48,...(40 无效)
4	延迟转移	延迟转移，无效状态	12,16,60,64,...(40 无效)
5	延迟转移，无效状态	任意控制转移指令	12,40,44,...(16 无效)

注：条件转移指令可根据判定成功与否，产生无转移和条件转移两种情况。带无效指令状态的指令，如果无效判定条件成立，会产生无效状态。

表 2.3　寄存器地址映射

全局寄存器和窗口寄存器地址	r 寄存器地址
in[0] - in[7]	r[24] - r[31]
local[0] - local[7]	r[16] - r[23]
out[0] - out[7]	r[8] - r[15]
global[0] - global[7]	r[0] - r[7]

多个窗口依次排列，每窗口与它相临的两个窗口共享 in 和 out 寄存器组，当前窗口的 in 寄存器是下一个窗口的 out 寄存器组，当前窗口的 out 寄存器组是上一个窗口的 in 寄存器组。图 2.2给出了 3 个窗口互相重叠时的情况。

多个窗口连成一个环。图 2.3给出由 8 个窗口组成的窗口寄存器环。在 SAVE 指令执行或陷阱产生时，当前窗口指针会指向下一个窗口所对应的 24 个窗口寄存器，即窗口发生旋转。这 24 个窗口寄存器和 8 个全局寄存器会组成新的 32 个通用寄存器。但由于窗口共享的特性，新窗口的 in 寄存器组是旧窗口的 out 寄存器组，旧窗口的 in、local 寄存器组被隐藏，不能被访问。同理，执行 RESTORE 和 RETT 指令引发窗口旋转后，上一个窗口的 in、local 寄存器组暴露出来。

SPARCv8 通过窗口机制、用空间换取时间、通过使用能多的寄存器、来提供更快速上下文切换和保存。在对窗口寄存器的建模中，同样需要与上述介绍等同数量的寄存器。如窗口数量为 8 时，我们需要 8*16 个窗口寄存器，即 128 个窗口寄存器；除此之外，我们还需要 8 个全局寄存器，共 136 个寄存器。但这

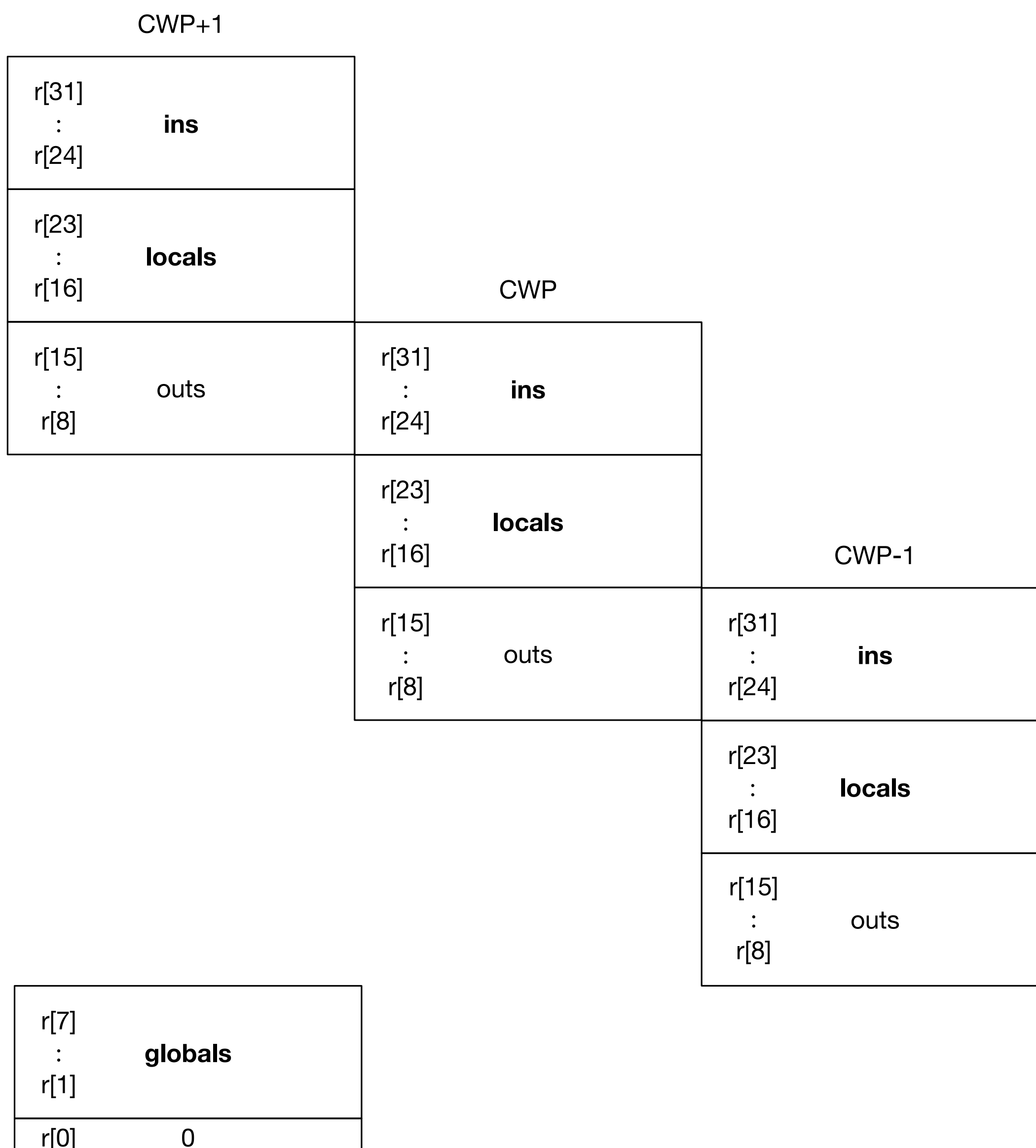


图 2.2 3 个互相重叠的窗口和 8 个全局寄存器

136 个寄存器在逻辑上并不是处于同一个层次，因为我们每次只能接触到其中的 32 个寄存器。考虑到更加清晰的展现窗口寄存器的特性，我们需要将其分为 32 个全局寄存器和 108 个剩余的窗口寄存器两部分。在本文中剩余的窗口寄存器用一个寄存器链表来表示，叫做"帧寄存器链表"。如何在这样的结构上刻画窗口旋转等操作也是在建模的操作语义中需要解决的问题。

2.1.5 陷阱

陷阱是指通过一个特殊的陷阱表，将控制权转移到管理模式的机制。陷阱表中包含每个陷阱处理函数的前 4 条指令。陷阱表的位置由管理模式决定。当陷阱

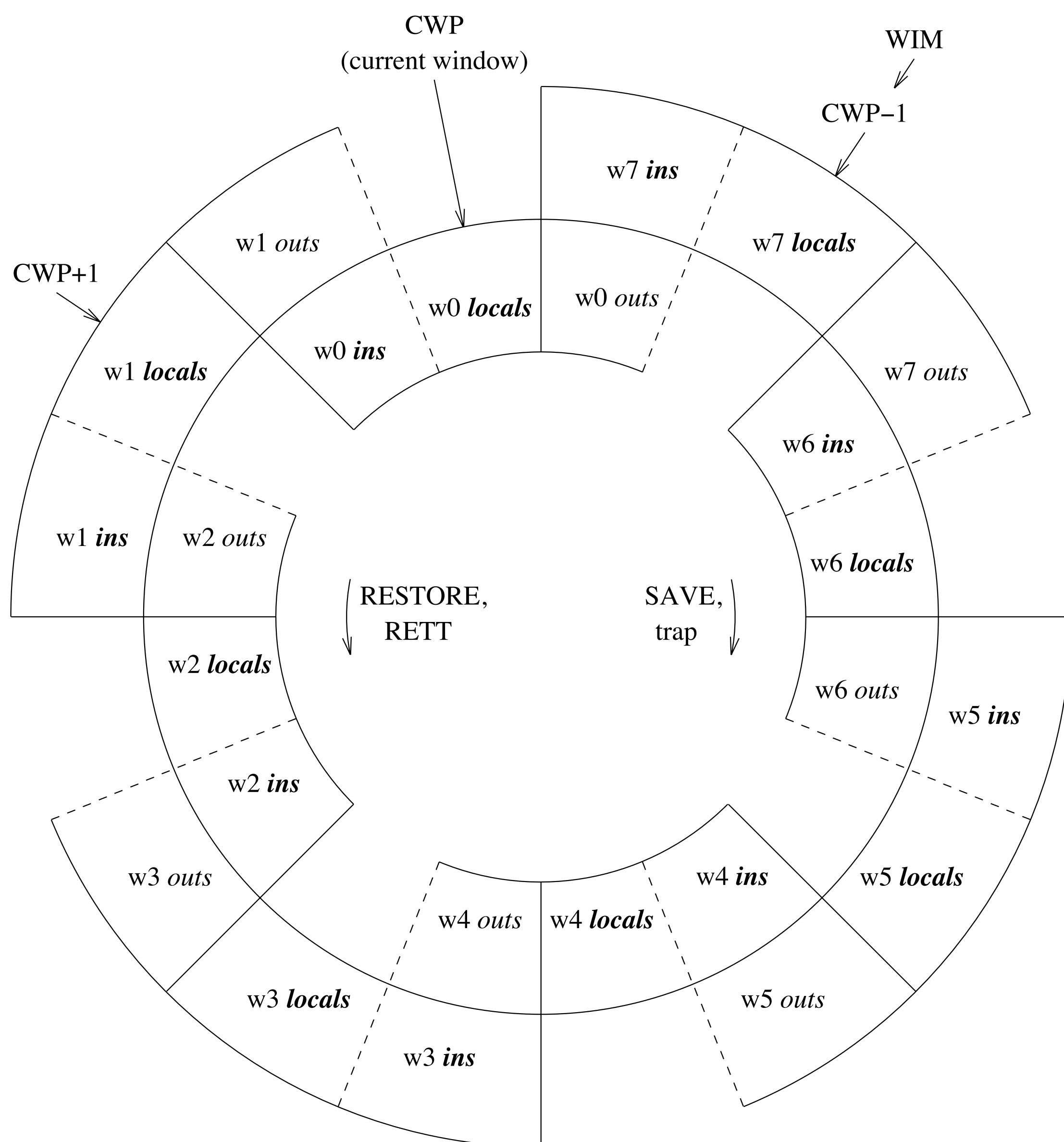


图 2.3　窗口数为 8 时的窗口寄存器

被触发时，会根据不同的陷阱类型，选择最高等级的陷阱，非延迟转移到陷阱表中此陷阱对应的处理函数并转换到管理模式。

SPARCv8 中的陷阱可分为 3 种，分别为软件陷阱，中断请求，指令异常。指令异常是指指令调度时，指令产生的除零，地址对齐等异常，优先级最高。软件陷阱是指用户通过 SPARC 汇编指令 Ticc 主动触发的陷阱，共 128 种，优先级次之。中断请求是指从主板传来的中断请求，共 15 种，优先级最低。在 SPARCv8 中，通过 PIL 寄存器来限制允许中断的等级。

2.1.6 模式

SPARCv8 中分为用户模式和管理模式。在访问存储器时，通过地址空间标识符（ASI）携带的模式信息，来读取不同的内存空间。因此用户模式和管理模式的内存空间具有一定的隔离性。除此之外，管理模式还可以控制，读写一些用户态没有访问权限的寄存器。

在建模中，为了刻画以上特性，我们需要 2 个内存空间，分别为用户模式和管理模式所使用。由于在系统运行时，所有的状态均是在某个模式下运行的，所以系统每时每刻只能访问到一个内存空间。故我们需要在系统每个指令周期初始时选出当前模式所对应的空间，而后再进行余下的操作。

2.1.7 写延迟

在 SPARCv8 中，状态寄存器的读写指令会对一些状态寄存器进行写入操作，这些状态寄存器包括 PSR 寄存器、TBR 寄存器、WIM 寄存器、ASR 寄存器、Y 寄存器。对这些寄存器的写操作会延迟 0 ～ 3 个周期执行，具体数值依赖于实现。除此之外，在这些延迟写入的状态寄存器中，寄存器的某些段可能会被立即写入。如写 PSR 指令执行后，会对 PSR 中的 ET 段和 PIL 段立即写入，其他位则延迟写入。

在建模过程中，写延迟的特性需要我们先将待写入状态寄存器的值保存起来，等到几个周期之后再将这些值存入具体的寄存器中。我们使用了一个链表来存储待写入状态寄存器的值和名称，以及需要等待的周期。这个链表被称为“延迟寄存器链表”。

2.1.8 指令集

SPARCv8 中提供了丰富的汇编指令来支持以上特性。指令可以被分为 4 种，分别为访存指令，整数运算指令，控制转移指令，状态寄存器的读写指令。

- 访存指令与内存的读写有关，如 LD、ST 指令，分别为对内存中 4 字节长度的空间进行读和写。
- 整数运算指令是指诸如加法（ADD）、减法（SUB）、按位与（AND）的指令。
- 控制转移指令主要有 6 种，分别为:

（a）执行分支的条件转移指令 Bicc。

（b）执行函数调用的延迟跳转指令 CALL。

（c）执行函数跳转的延迟跳转指令 JMPL。

（d）用于陷阱返回的延迟跳转指令 RETT。

（e）触发软件中断的指令 Ticc。

（f）用于保存和恢复窗口的 SAVE 指令和 RESTORE 指令。

- 状态寄存器的读写指令是指对状态寄存器的延迟写操作的指令。

2.1.9 本节小结

综上所述，为了描述 SPARCv8 中以上的特性，我们需要：

- PC 寄存器、nPC 寄存器和无效指令状态寄存器，来刻画控制转移特性。
- 32 个通用寄存器、帧寄存器链表、表示当前窗口指针的 CWP 寄存器、表示窗口是否可用的 WIM 寄存器来刻画窗口特性。
- 表示陷阱表位置的 TBA 寄存器、表示陷阱类型的 tt 寄存器、表示允许中断等级的 PIL 寄存器、表示当前是否处于陷阱状态的陷阱状态寄存器，以及表示是否允许陷阱发生的 ET 寄存器来刻画陷阱特性。
- 表示当前模式的 S 寄存器、表示之前模式的 PS 寄存器来刻画模式的特性。
- 延迟寄存器链表来刻画写延迟特性。

除此之外，完整描述 SPARCv8 还需要：

- 与计算条件判断有关的 n、z、v、c 寄存器，分别表示是否为负数、是否为零、是否溢出、是否进位。与除法计算相关的 y 寄存器。
- 用于存储处理器状态的辅助状态寄存器组 ASR。

注意到，状态寄存器的读写指令中 WIM、ASR、Y 寄存器均在上述给出的寄存器中，而 PSR、TBR 寄存器不在。原因是由于 PSR 寄存器、TBR 寄存器中不段有不同的意义，故在建模中对这些寄存器进行了拆分。

我们将 PSR 寄存器，TBR 寄存器的不同段拆开，并且不考虑与建模无关的段（如 PSR 寄存器中实现有关的 impl 段、与版本有关的 ver 段、保留的 reserved

段、是否开启协处理器的 EC 段、是否开启浮点寄存器的 EF 段，以及 TBR 寄存器的零段）。

2.2 证明工具 Coq 简介

Coq 是法国国家科学研究中心等机构研发的一个交互式的定理证明辅助工具。它提供提供了一套规范语言，允许用户形式化的定义数学表达式、程序语言、算法等等，并可以基于这些定义构造形式化的证明。Coq 在近几年中被广泛应用在系统软件的形式化验证中 [15, 28, 31, 32]。本文所描述的 SPARCv8 的形式化建模以及相关证明工作全部在 Coq 中实现。

2.2.1 归纳定义

在 Coq 中, "Inductive"用来归纳定义一类型。归纳定义需要一个名称、一个类型、零个或多个规则来定义一个归纳类，每个规则都有一个独特的名称，称之为"构造函数"。

代码2.3为自然数的定义，其中 O 为一个自然数，S 是一个构造函数，它需要一个自然数作为输入，输出则为另一个自然数。也就是说，如果 n 是一个自然数，那么 S n 亦为一个自然数。

代码2.4为小于等于的定义，这个归纳定义需要一个自然数 n 作为输入，输出为 nat → Prop, 即从自然数映射到 Coq 中逻辑命题的一个函数。这个定义包含 2 条规则，第一条规则声明任何 n 小于等于它自身。第二条规则声明对于任意自然数 m，如果满足 $n \leq m$，则一定有 $n \leq m+1$。

代码 2.3 在 Coq 中定义自然数

```
1 Inductive nat: Type :=
2   | O: nat
3   | S: nat -> nat.
```

代码 2.4 在 Coq 中定义自然数小于等于判断

```
1  Inductive le (n: nat): nat -> Prop :=
2    | le_n: le n n
3    | le_S: forall m:nat, le n m -> le n (S m).
```

代码 2.5 在 Coq 中定义自然数前驱

```
1 Definition pred (n : nat) : nat :=
2  match n with
3   | O => O
```

```
4   | S n’  => n’
5 end.
```

代码 2.6　在 Coq 中定义自然数相加

```
1 Fixpoint plus (n: nat) (m: nat) : nat :=
2   match n with
3   | O => m
4   | S n’ => S (plus n’ m)
5 end.
```

2.2.2　定义非递归函数和递归函数

在 Coq 中,“Definition”和“Fixpoint”分别用来定义非递归函数和递归函数。代码2.5给出了使用“Definition”定义的自然数前驱函数。

在 Coq 中递归函数的定义会强制要求函数的某个参数是结构上递减的，因此在语法程度上保证了函数的中止。如代码2.6为使用“Fixpoint”定义的自然数加法函数，当 Coq 检查这个定义时，会认为第一个参数是递减的，也就是说我们在参数 n 上进行了结构递归。我们每次调用这个递归函数，参数 n 总会变的更小，所以 plus 函数终会中止。

2.2.3　证明对象和证明脚本

证明对象（proof object）是 Coq 的核心组成部分。当 Coq 运行证明脚本（proof script）时，会逐渐构造起一个证明项，证明项的类型就是需要证明的命题（proposition）。“Proof”和“Qed”之间的证明策略告诉我们怎样去构造一个这样的证明项。

代码 2.7　证明 0 小于等于 2

```
1  Theorem le_0_2:le 0 2.
2  Proof.
3    (* le 0 2 *)
4    apply le_S.
5
6    (* le 0 1 *)
7    apply le_S.
8
9    (* le 0 0 *)
10   apply le_n.
11 Qed.
```

在任意时刻，Coq 构建证明项时都伴随着一个证明目标，每个证明目标都对应于若干个子目标，如果证明中没有了子目标，则证明完毕。代码2.6给出了 0 小

于等于 2（le 0 2）的证明。在第一次应用“le_S”后，证明子目标变为“le 0 1”，再次应用“le_S”后，证明子目标变为“le 0 0”。此时应用“le_n”，不再产生新的子目标，则证明完毕。

第 3 章　SPARCv8 指令集的形式化建模

SPARCv8 指令集合为程序员提供了一个面向硬件的汇编编程语言。对某种编程语言的形式化建模，首先需要给出语言的抽象语法定义，然后定义抽象的机器状态，最后给出语言每条语句在机器状态上对应的操作语义，即指令的行为。本章从这三个方面出发，对 SPARCv8 指令集的进行建模。在第一节中，给出了 SPARCv8 指令集合对应的抽象语法，在第二节中，定义了 SPARCv8 指令执行的机器状态，在第三节给出其操作语义。

3.1　汇编指令语法

本节主要介绍 SPARCv8 汇编指令的抽象语法，如图 3.1所示。

$$
\begin{array}{rrcl}
(Word) & w & \in & Int32 \\
(GenReg) & r & ::= & r_0 \mid \ldots \mid r_{31} \\
(AsReg) & asr & ::= & asr_0 \mid \ldots \mid asr_{31} \\
(Symbol) & \varsigma & ::= & psr \mid wim \mid tbr \mid y \mid asr \\
(OpExp) & \alpha & ::= & r \mid w \\
(AddrExp) & \beta & ::= & \alpha \mid r+\alpha \\
(TrapExp) & \gamma & ::= & r \mid r+r \mid r+w \mid w \\
(TestCond) & \eta & ::= & al \mid nv \mid ne \mid eq \mid \ldots \\
(SparcIns) & i & ::= & \mathbf{ticc}\,\eta\,\gamma \mid \mathbf{rett}\,\beta \mid \mathbf{save}\,r_s\,\alpha\,r_d \mid \mathbf{restore}\,r_s\,\alpha\,r_d \mid \\
 & & & \mathbf{ld}\,\beta\,r_d \mid \mathbf{st}\,r_d\,\beta \mid \mathbf{rd}\,\varsigma\,r_d \mid \mathbf{wr}\,r_d\,\alpha\,\varsigma \mid \\
 & & & \mathbf{bicc}\,\eta\,\beta \mid \mathbf{bicca}\,\eta\,\beta \mid \mathbf{jmpl}\,\beta\,r_d \mid \mathbf{nop} \mid \\
 & & & \mathbf{udivcc}\,r_s\,\alpha\,r_d \mid \ldots
\end{array}
$$

图 3.1　SPARCv8 汇编指令的抽象语法

w 代表 32 位的机器数（$Word$）。asr 代表 32 个辅助寄存器（$AsReg$），名称用 asr_0 至 asr_{31} 表示。ς 为标志寄存器（$Symbol$）的名称，包括 PSR 寄存器 (psr)、WIM 寄存器 (wim)、TBR 寄存器 (tbr)、Y 寄存器 (y)、ASR 寄存器 (asr)，这些寄存器具体的含义可见上一章节。

r 代表 32 个通用寄存器（$GenReg$），名称用 r_0 至 r_{31} 表示。32 个通用寄存器中，r_0 至 r_{31} 还分别对应一些别名，如表 3.1所示。需要注意的是，32 个通用寄存器中，$r_0(g_0)$ 寄存器是较为特殊的寄存器，它的值恒为 0，所以对此寄存器

表 3.1　通用寄存器及对应别名

通用寄存器	别名
r0 - r7	g0 - g7
r8 - r15	o0 - o7
r16 - r23	l0 - l7
r24 - r31	i0 - i7
r14	sp
r30	fp

注：sp 即栈指针（stack pointer），fp 即帧指针（frame pointer）。

的读操作返回值为 0，写操作则对此寄存器不产生影响，即：

$$R\{r \rightsquigarrow w\} \stackrel{def}{=\!=} \begin{cases} R & \textbf{if } r = r_0 \\ R[r \rightsquigarrow w] & \textbf{otherwise} \end{cases}$$

$$[\![r]\!]_R = \begin{cases} 0 & \textbf{if } r = r_0 \\ R(r) & \textbf{otherwise} \end{cases}$$

α、β、γ 分别代表了操作数表达式（$OpExp$）、地址表达式（$AddrExp$）和陷阱表达式（$TrapExp$）。三者语法类似，均由立即数或通用寄存器通过一定运算组成。操作数表达式可以是立即数或通用寄存器。地址表达式有 4 种，分别为立即数，通用寄存器，通用寄存器与通用寄存器相加，通用寄存器与立即数相加。陷阱表达式的类型与地址表达式的类型一致。

虽然陷阱表达式与地址表达式的类型一致，但却无法像地址表达式一样复用操作数表达式的定义。原因是 SPARC 指令中，不同的汇编指令中传入的立即数有不同的取值范围。由于指令宽度固定为 32 位，除去其他操作数，留给指令传入的立即数的长度必然是有限的。如 LD、ST 等指令中，地址表达式传入的立即数为有符号 13 位机器数，数值应在-4096 至 4095 之间。TICC 指令中陷阱表达式传入的陷阱标号是无符号或有符号 7 位机器数，数值在-64 至 63 或者 0 至 127 之间。操作数表达式中立即数为有符号 13 位机器数，故地址表达式可以复用其定义。

为了使模型语法更简单，三者对于立即数范围的限制并没有在语法部分给

出，而是在表达式求值时再做判断：

$$[\![\alpha]\!]_R = \begin{cases} [\![r_m]\!]_R & \textbf{if } \alpha = r_m \\ w & \textbf{if } \alpha = w, -4096 \le w \le 4095 \\ \bot & \textbf{otherwise} \end{cases}$$

$$[\![\beta]\!]_R = \begin{cases} [\![\alpha]\!]_R & \textbf{if } \beta = \alpha \\ [\![r_m]\!]_R + [\![\alpha]\!]_R & \textbf{if } \beta = r + \alpha \end{cases}$$

$$[\![\gamma]\!]_R = \begin{cases} [\![r_m]\!]_R & \textbf{if } \gamma = r_m \\ [\![r_m]\!]_R + [\![r_n]\!]_R & \textbf{if } \gamma = r_m + r_n \\ [\![r_m]\!]_R + w & \textbf{if } \gamma = r_m + w, -64 \le w \le 63 \\ w & \textbf{if } \gamma = w, 0 \le w \le 127 \\ \bot & \textbf{otherwise} \end{cases}$$

表达式求值时，如果不满足数值范围条件，则返回空值。

η 代表了条件判断表达式（$TestCond$），由恒成立（al）、恒不成立（nv）、不相等（ne）、相等（eq）等判断条件组成，**ticc** 指令和 **bicc** 指令中需要一个条件判断表达式，即条件判断表达式与 **ticc** 指令或 **bicc** 指令共同确定一条 SPARC 汇编指令。**ticc** al 指令为原语法中的 TA 指令，**ticc** nv 为原语法中的 TN 指令。**bicca** 指令为存在无效指令状态判断的指令。如 **bicca** al 对应于原语法中的 BA,a 指令。条件表达式求值时，需要根据条件种类（如相等、小于等等），通过读取条件寄存器（n、z、v、c）的值，来判断条件是否成立：

$$[\![\eta]\!]_R = \begin{cases} true & \textbf{if } \eta = al \\ false & \textbf{if } \eta = nv \\ \textbf{if } (R(z)) = 0) \textbf{ then } true \textbf{ else } false & \textbf{if } \eta = ne \\ \textbf{if } (R(z)) \neq 0) \textbf{ then } true \textbf{ else } false & \textbf{if } \eta = eq \\ \ldots & \ldots \end{cases}$$

i 代表 SPARC 汇编指令（$SparcIns$），汇编指令后面会有若干个参数。如 **ticc** 指令需要条件判断表达式和陷阱表达式作为参数，**rett** 指令需要地址表达式作为参数，**save** 指令需要指定 r_s 寄存器和 r_d 寄存器以及一个操作数表达式。

被省略的指令均为算术运算指令，在此不一一列出。

3.2 机器状态相关定义

为了定义指令对应的操作语义，即指令在机器状态上执行的行为，需要先定义机器状态。本节主要介绍了形式化建模中所定义的机器状态。在第一小节中，给出了寄存器文件的相关定义。在第二小节中，给出了帧寄存器链表及相关操作

的定义。在第三小节中，给出了延迟寄存器链表及相关操作的定义。在第四小节中，对一上三小节内容进行总结，给出了机器状态的定义。同时也给出了内存模型和代码堆的定义。

3.2.1 寄存器文件

$$
\begin{array}{rrcl}
(PsrSeg) & \mu & ::= & n \mid z \mid v \mid c \mid pil \mid s \mid ps \mid et \mid cwp \\
(TbrSeg) & \nu & ::= & tba \mid tt \\
(RegName) & q & ::= & r \mid \mu \mid \nu \mid wim \mid y \mid asr \mid pc \mid npc \mid \kappa \mid \tau \\
(RegFile) & R & \in & RegName \rightarrow Word
\end{array}
$$

图 3.2 寄存器文件的定义

图 3.2给出了寄存器文件的定义。

μ，ν 分别代表 PSR 寄存器段（$PsrSeg$）和 TBR 寄存器段（$TbrSeg$），在 SPARCv8 真实机器中，PSR、TBR 寄存器是 32 位寄存器，32 位中每个位代表不同的含义，如 PSR 寄存器中 ET 段表示当前状态是否允许陷阱产生，0 表示不允许，1 表示允许。PS、S 段表示当前和之前机器所处的模式，0 表示用户模式，1 表示管理者模式。n、z、v、c 段为条件寄存器，用于相等，小于等条件判断时使用。PIL 段用于表示当前允许被中断的等级，CWP 段表示当前窗口标号。TBR 寄存器中 TBA 段表示陷阱表的地址，tt 段表示陷阱种类。考虑到具体实现的限制，SPARCv8 真实机器中将这些功能迥然的标志位放入同一个寄存器中。但在建模中不受具体实现的限制，为了更清楚的表现寄存器的状态，我们将以上标志位拆分开。拆分后，每个段寄存器中不同的值对应于不同的含义，比如：

$$
\begin{array}{lcllcl}
\mathsf{trap_disabled}(R) & \stackrel{def}{=\!=} & R(et) = 0 & \mathsf{usr_mode}(R) & \stackrel{def}{=\!=} & R(s) = 0 \\
\mathsf{trap_enabled}(R) & \stackrel{def}{=\!=} & R(et) \neq 0 & \mathsf{sup_mode}(R) & \stackrel{def}{=\!=} & R(s) \neq 0
\end{array}
$$

et 寄存器值为 0 时，表示机器当前处于关陷阱 (trap_disabled) 状态，et 寄存器值不为 0 时，表示机器当前处于开陷阱 (trap_enabled) 状态。s 寄存器值为 0 时，表示机器当前处于用户模式（usr_mode），s 寄存器值不为 0 时，表示机器当前处于管理者模式（sup_mode）。

除了具体的状态，我们还可以定义一些对这些寄存器操作的函数：

$$
\begin{array}{lcllcl}
\mathsf{to_usr}(R) & \stackrel{def}{=\!=} & R\{s \rightsquigarrow 0\} & \mathsf{to_sup}(R) & \stackrel{def}{=\!=} & R\{s \rightsquigarrow 1\} \\
\mathsf{save_mode}(R) & \stackrel{def}{=\!=} & R\{ps \rightsquigarrow R(s)\} & \mathsf{restore_mode}(R) & \stackrel{def}{=\!=} & R\{s \rightsquigarrow R(ps)\} \\
\mathsf{enable_trap}(R) & \stackrel{def}{=\!=} & R\{et \rightsquigarrow 1\} & \mathsf{disable_trap}(R) & \stackrel{def}{=\!=} & R\{et \rightsquigarrow 0\}
\end{array}
$$

如用户模式和管理者模式之间的切换（to_usr、to_sup），保存模式（save_mode），即将寄存器 s 中的值保存到寄存器 ps 中，恢复模式（restore_mode），即将寄存器 ps 中的值恢复到寄存器 s 中，开关陷阱（enable_trap，disable_trap）等等。

q 代表寄存器名称（$RegName$），包括通用寄存器、部分标志寄存器（wim，y，asr）以及 PSR 寄存器段（psr）和 TBR 寄存器段（tbr）。还有表示程序跳转的 pc，npc 寄存器和用来表示陷阱状态和无效指令状态的 κ、τ 寄存器。这些寄存器全映射到 32 位的机器数，则组成了寄存器文件（$RegFile$）。我们给出一些关于 κ、τ 寄存器状态的定义，如：

$$\text{annuled}(R) \stackrel{def}{=\!=} R(\kappa) \neq 0 \qquad \text{has_trap}(R) \stackrel{def}{=\!=} R(\tau) \neq 0$$

κ 寄存器值为不为 0 时，表示机器当前处于无效指令状态。τ 寄存器值不为 0 时，表示机器当前处于陷阱（has_trap）状态。还有一些对 κ，τ 寄存器操作的函数：

$$\begin{array}{llll} \text{set_annul}(R) \stackrel{def}{=\!=} R\{\kappa \rightsquigarrow 1\} & \qquad \text{set_trap}(R) \stackrel{def}{=\!=} R\{\tau \rightsquigarrow 1\} \\ \text{clear_annul}(R) \stackrel{def}{=\!=} R\{\kappa \rightsquigarrow 0\} & \qquad \text{clear_trap}(R) \stackrel{def}{=\!=} R\{\tau \rightsquigarrow 0\} \end{array}$$

如设置无效指令状态（set_annul）和清除无效指令状态（clear_annul），设置陷阱状态（set_trap）和清清除陷阱状态（clear_trap）。

同时，我们还可以定义作用在 pc、npc 寄存器上的控制转移函数：

$$\begin{array}{ll} \text{next}(R) & \stackrel{def}{=\!=} R\{pc \rightsquigarrow R(npc)\}\{npc \rightsquigarrow R(npc)+4\} \\ \text{djmp}(w, R) & \stackrel{def}{=\!=} R\{pc \rightsquigarrow R(npc)\}\{npc \rightsquigarrow w\} \\ \text{tbr_jmp}(R) & \stackrel{def}{=\!=} R\{pc \rightsquigarrow R(tbr)\}\{npc \rightsquigarrow R(tbr)+4\} \end{array}$$

控制转移类指令可分为 4 种，分别为无转移指令、条件延迟转移指令、延迟转移指令、非延迟转移指令。条件延迟转移指令可以根据条件判断的结果，转化为无转移指令或延迟转移指令。所以只需要无转移、延迟转移、非延迟转移 3 个函数即可，其中非延迟转移指令只在执行陷阱时发生，此时新的地址固定为 TBR 的值。

由于 psr 和 tbr 寄存器在本模型中各段是分离的，故在读取时需要将相应段重新组合成一个 32 位机器数。即：

$$\begin{array}{l} R(psr) \stackrel{def}{=\!=} w \\ \quad \textbf{where} \quad w_{<31:24>} = 0, w_{<23>} = R(n), w_{<22>} = R(z), w_{<21>} = R(v), \\ \qquad\qquad w_{<20>} = R(c), w_{<19:12>} = 0, w_{<11:8>} = R(pil), w_{<7>} = R(s), \\ \qquad\qquad w_{<6>} = R(ps), w_{<5>} = R(et), w_{<4:0>} = R(cwp) \\ R(tbr) \stackrel{def}{=\!=} w \\ \quad \textbf{where} \quad w_{<31:12>} = tba, w_{<11:4>} = tt, w_{<3:0>} = 0 \end{array}$$

3.2.2 帧寄存器链表

为了刻画窗口旋转的特性，我们使用 32 个通用寄存器和 2N-3 个大小为 8 的帧寄存器组（$Frame$）所组成帧寄存器链表（$FrameList$）来表示 SPARCv8 中的全局寄存器和窗口寄存器（其中 N 为窗口个数）。通用寄存器的定义已在上一节中给出，帧寄存器组和帧寄存器链表的定义如图 3.3所示。

$$
\begin{array}{rl}
(Frame)\ f & ::= [w_0, \ldots, w_7] \\
(FrameList)\ F & ::= \textbf{nil} \mid f :: F
\end{array}
$$

图 3.3 帧寄存器链表的定义

32 个通用寄存器代表了当前能够访问到的全局寄存器和窗口寄存器。帧寄存器链表（$FrameList$）中的寄存器则代表了当前无法访问到的窗口寄存器。

寄存器文件和帧寄存器链表组成的状态称为 R 状态（$RState$）:

$$(RState)\ Q ::= (R, F)$$

一些定义在寄存器文件上的操作同样可以定义在 R 状态上:

$$
\begin{array}{lcll}
\mathsf{has_trap}(Q) & \overset{def}{=\!=} & \mathsf{has_trap}(R) & \textbf{where } Q = (R, F) \\
\mathsf{usr_mode}(Q) & \overset{def}{=\!=} & \mathsf{usr_mode}(R) & \textbf{where } Q = (R, F) \\
\mathsf{sup_mode}(Q) & \overset{def}{=\!=} & \mathsf{sup_mode}(R) & \textbf{where } Q = (R, F) \\
\mathsf{annuled}(Q) & \overset{def}{=\!=} & \mathsf{annuled}(R) & \textbf{where } Q = (R, F)
\end{array}
$$

窗口的旋转会有通用寄存器和帧寄存器链表之间的数据交换，所以很多关于窗口旋转的定义需要定义在 R 状态上。当窗口需要旋转 1 次时,需要进行如下 3 个操作:

1. 首先将 32 个通用寄存器中 out、local、in 寄存器组转换为 3 个帧寄存器组:

$$
\begin{array}{lcl}
R[r_i, \ldots, r_{i+7}] & \overset{def}{=\!=} & [R(r_i), \ldots, R(r_{i+7})] \\
R\{[r_i, \ldots, r_{i+7}] \rightsquigarrow f\} & \overset{def}{=\!=} & R\{r_i \rightsquigarrow w_0\} \ldots \{r_{i+7} \rightsquigarrow w_7\} \\
 & & \textbf{where } f = [w_0, \ldots, w_7]
\end{array}
$$

$$\mathsf{fetch}(R) \overset{def}{=\!=} R[r_8, \ldots, r_{15}] :: R[r_{16}, \ldots, r_{23}] :: R[r_{24}, \ldots, r_{31}] :: \textbf{nil}$$

2. 将这 3 个帧寄存器组并入帧寄存器链表的尾端，再将帧寄存器链表旋转 2 次。其中帧寄存器链表旋转分为左旋和右旋。帧寄存器链表左旋时，将帧

寄存器链表头部的一个帧寄存器组分离，并入帧寄存器链表尾部。帧寄存器链表右旋时，将帧寄存器链表尾部的一个帧寄存器组分离，并入帧寄存器链表头部：

$$\mathsf{left}(F_L, F_l) \overset{def}{=\!=} (\, F_L' \; ++ \, (p :: q :: \mathbf{nil})\, ,\; F_l' \; ++ \, (m :: n :: \mathbf{nil})\,) \quad \mathbf{where}\ F_L = m :: n :: F_L', F_l = p :: q :: F_l'$$

$$\mathsf{right}(F_L, F_l) \overset{def}{=\!=} (\, (p :: q :: \mathbf{nil}) \; ++ \, (F_L')\, ,\; (m :: n :: \mathbf{nil}) \; ++ \, (F_l')\,) \quad \mathbf{where}\ F_L = F_L' ++ (m :: n :: \mathbf{nil})\, ,\; F_l = F_l' ++ (p :: q :: \mathbf{nil})$$

$$\mathsf{left_iter}(k, F_L, F_l) \overset{def}{=\!=} \begin{cases} (F_L, F_l) & \mathbf{if}\ k = 0 \\ \mathsf{left_iter}(k-1, \mathsf{left}(F_L, F_l)) & \mathbf{otherwise} \end{cases}$$

$$\mathsf{right_iter}(k, F_L, F_l) \overset{def}{=\!=} \begin{cases} (F_L, F_l) & \mathbf{if}\ k = 0 \\ \mathsf{right_iter}(k-1, \mathsf{right}(F_L, F_l)) & \mathbf{otherwise} \end{cases}$$

3. 最后，将旋转过的帧寄存器链表后 3 个帧寄存器组分离，写入 32 个通用寄存器中对应位置。

$$\mathsf{replace}(l, R) \overset{def}{=\!=} R\{[r_8, \ldots, r_{15}] \rightsquigarrow f_o\}\{[r_{16}, \ldots, r_{24}] \rightsquigarrow f_l\}\{[r_{24}, \ldots, r_{31}] \rightsquigarrow f_i\} \quad \mathbf{where}\ l = f_o :: f_l :: f_i :: \mathbf{nil}$$

图 3.4 给出了窗口左旋 1 次时，通用寄存器组和帧寄存器链表的变化情况。

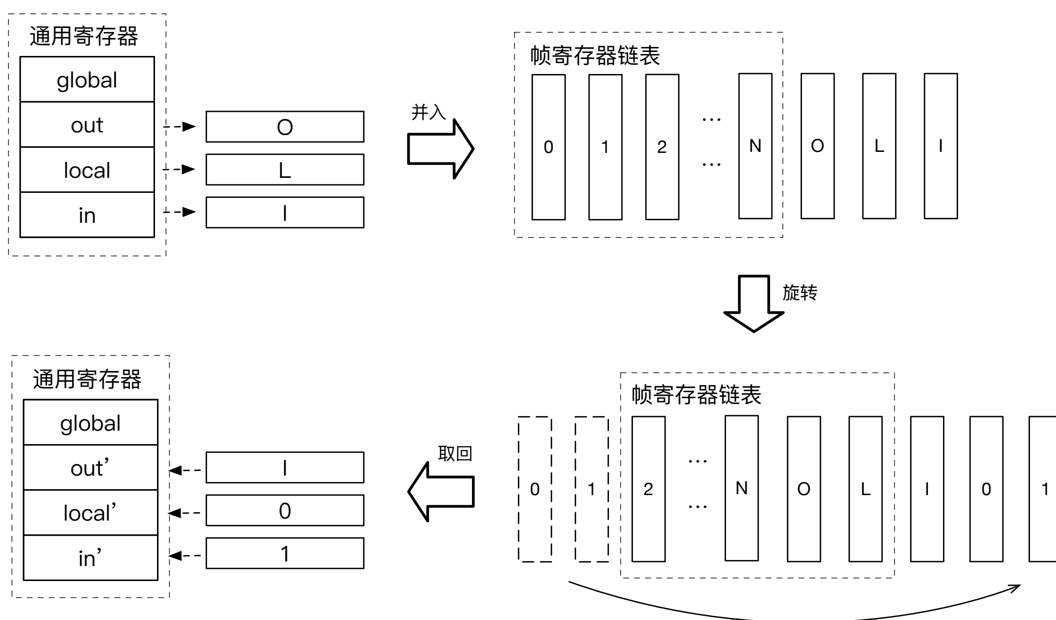


图 3.4 例：窗口左旋 1 次

帧寄存器链表旋转与当前窗口指针（CWP）值的修改是绑定操作的。CWP 值模增 1 时，窗口左旋 1 次。CWP 值模减 1 时，窗口右旋 1 次。如果对 CWP 值

进行赋值操作时，需要先根据旧的和新的 CWP 值之间的差值的大小，旋转相应的次数，再将新的 CWP 值赋值给 CWP 寄存器。所以窗口的左旋和右旋操作除了以上 3 个步骤，还需要在最后修改 CWP 的值：

$$\begin{aligned}\mathsf{left_win}(w, Q) &\overset{def}{=\!=} \textbf{let } (F', l) ::= \mathsf{left_iter}(w, F, \mathsf{fetch}(R)) \textbf{ in} \\ &\qquad \textbf{let } R' ::= \mathsf{replace}(l, R) \textbf{ in } (R'\{cwp \rightsquigarrow \underline{\mathsf{post_cwp}(w, R)}\}, F') \\ &\textbf{where } Q = (R, F)\end{aligned}$$

$$\begin{aligned}\mathsf{right_win}(w, Q) &\overset{def}{=\!=} \textbf{let } (F', l) ::= \mathsf{right_iter}(w, F, \mathsf{fetch}(R)) \textbf{ in} \\ &\qquad \textbf{let } R' ::= \mathsf{replace}(l, R) \textbf{ in } (R'\{cwp \rightsquigarrow \underline{\mathsf{pre_cwp}(w, R)}\}, F') \\ &\textbf{where } Q = (R, F)\end{aligned}$$

其中，pre_cwp，post_cwp 为当前窗口的上一个窗口和下一个窗口：

$$\mathsf{pre_cwp}(w, R) \overset{def}{=\!=} (R(cwp) - w + N) \textbf{ mod } N$$

$$\mathsf{post_cwp}(w, R) \overset{def}{=\!=} (R(cwp) + w) \textbf{ mod } N$$

在 SPARC 汇编指令中，SAVE、RESTORE 等指令对窗口的旋转操作 inc_windec_win 需要先判断当前窗口是否是可用的，WR 指令写 PSR 寄存器时对于窗口的修改 set_win 操作则不需要判断：

$$\mathsf{set_win}(w, Q) \overset{def}{=\!=} \begin{cases} \mathsf{left_win}(w - R(cwp), Q) & \textbf{if } w > R(cwp) \\ \mathsf{right_win}(R(cwp) - w, Q) & \textbf{if } w < R(cwp) \\ Q & \textbf{otherwise} \end{cases}$$
$$\textbf{where } Q = (R, F)$$

$$\mathsf{inc_win}(Q) \overset{def}{=\!=} \begin{cases} \mathsf{left_win}(1, Q) & \textbf{if } \neg\mathsf{win_masked}(\mathsf{post_cwp}(1, R), R) \\ \bot & \textbf{otherwise} \end{cases}$$
$$\textbf{where } Q = (R, F)$$

$$\mathsf{dec_win}(Q) \overset{def}{=\!=} \begin{cases} \mathsf{right_win}(1, Q) & \textbf{if } \neg\mathsf{win_masked}(\mathsf{pre_cwp}(1, R), R) \\ \bot & \textbf{otherwise} \end{cases}$$
$$\textbf{where } Q = (R, F)$$

其中

$$\mathsf{win_masked}(w, R) \overset{def}{=\!=} 2^w \&\& R(wim) \neq 0$$

我们可以看出，窗口左旋（left_win）与窗口标号增加 inc_win，即指向下一个窗口之间的区别就是后者有对当前窗口是否可用的检验。同理窗口右旋与窗口标号减小之间也存在类似关系。

3.2.3 延迟寄存器链表

为了刻画写延迟特性，我们使用一个延迟寄存器链表（$DelayList$）来存储写延迟相关的信息，其中存储的元素（$DelayItem$）为延迟周期（$DelayCycle$），

寄存器名称，需要延迟写入的值组成的 3 元组，如图 3.5所示。

$$
\begin{array}{rrcl}
(DelayCycle) & c & \in & \{0,1,\ldots,X\} \\
(DelayItem) & d & ::= & (c,\varsigma,w) \\
(DelayList) & D & ::= & \textbf{nil} \mid d :: D
\end{array}
$$

图 3.5　延迟寄存器链表的定义

其中延迟周期在 0 和 X 之间，X 的值依赖于具体实现。涉及到写延迟的操作有 2 个，分别为将三元组插入延迟寄存器链表的操作和延迟寄存器链表的例行检查操作，具体操作语义将在第三章给出。

3.2.4　机器状态相关定义

图 3.6给出了机器状态相关定义。

$$
\begin{array}{rrcl}
(Address) & a & \in & Word \\
(Memory) & M & \in & Address \rightharpoonup Word \\
(MemPair) & \Phi & ::= & (M_u, M_s) \\
(State) & S & ::= & (\Phi, Q, D)
\end{array}
$$

图 3.6　机器状态的定义

M 代表内存（$Memory$），为地址（$Address$）到 32 位的机器数的部分映射，其中地址亦为 32 位的机器数。用户模式的内存和管理者模式的内存共同组成了内存组（$MemPair$），用 Φ 表示。

内存组，R 状态和延迟寄存器链表确定了机器的状态（$State$）。一些定义在 R 状态上的操作同样可以定义在机器状态上：

$$
\begin{array}{lcll}
\mathsf{has_trap}(S) & \stackrel{def}{=\!=} & \mathsf{has_trap}(Q) & \textbf{where } S=(\Phi,Q) \\
\mathsf{usr_mode}(S) & \stackrel{def}{=\!=} & \mathsf{usr_mode}(Q) & \textbf{where } S=(\Phi,Q) \\
\mathsf{sup_mode}(S) & \stackrel{def}{=\!=} & \mathsf{sup_mode}(Q) & \textbf{where } S=(\Phi,Q)
\end{array}
$$

除了机器状态以外，我们还给出了指令堆（$CodeHeap$）、指令堆组（$CodePair$）、事件（$Event$）的定义。如图 3.7所示。

C 代表指令堆，为标号（$Label$）到汇编指令（$SparcIns$）的部分映射，其中标号为 32 位的机器数。用户模式的指令堆和管理者模式的指令堆共同组成了指令堆组，用 Δ 表示。机器的状态和指令堆组共同构成整个机器（$World$）。e 代

$$
\begin{array}{rlll}
(Label) & l & \in & Word \\
(CodeHeap) & C & \in & Label \rightharpoonup SparcIns \\
(CodePair) & \Delta & ::= & (C_u, C_s) \\
(World) & W & ::= & (\Delta, S) \\
(Event) & e & ::= & w \mid \bot \\
(EventList) & E & ::= & \mathbf{nil} \mid e :: E
\end{array}
$$

图 3.7　代码堆和事件的定义

表事件，如果有陷阱产生并被执行，则记录陷阱标号作为事件。多步执行时产生的多个陷阱标号组成事件序列（$EventList$）。

3.3　操作语义

在操作语义的定义中，我们首先定义一些作用于寄存器文件和内存上的汇编指令的操作语义，这些汇编指令不涉及到窗口旋转，延迟写等特性。而后，我们再定义一些作用于寄存器文件，内存，帧寄存器链表和延迟寄存器链表的的汇编指令的操作语义，这些汇编指令涉及到窗口旋转，写延迟指令故需要帧寄存器链表和延迟寄存器链表进行相应操作。而后，我们加入无效指令状态转移和延迟写状态转移的操作语义，来体现这两个特性。最后，我们加入中断，陷阱执行和模式选择的操作语义，最终成为单步执行的操作语义，来体现这 3 个特性。整体结构如图 3.8所示。

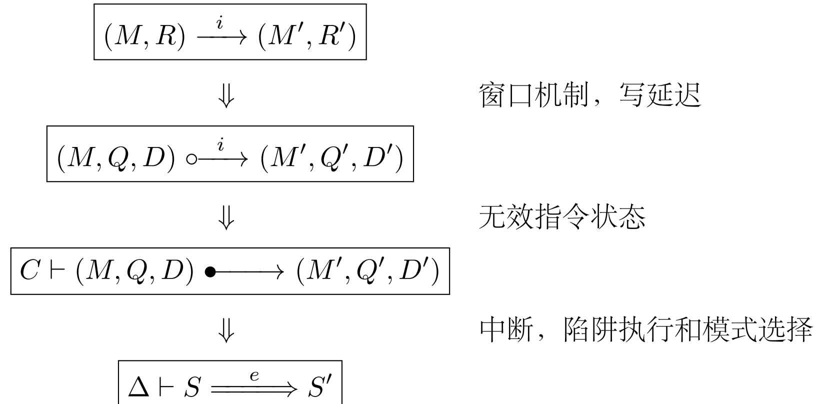


图 3.8　操作语义建模的结构

在本章中，第一小节给出了由单步执行合成多步执行的操作语义。第二小节给出了单步执行中首先执行的中断处理，陷阱执行和模式选择。第三小节给出了单步执行中随后执行的延迟状态转移和无效指令状态的转移。第四小节给出了

指令调度的操作语义。第五小节介绍了指令异常及处理方式。

3.3.1 单步执行和多步执行

在一个指令周期内，系统状态的变化的规则为：$\Delta \vdash S \xRightarrow{e} S'$。系统从状态 S 变化为状态 S'，在状态变化的同时产生事件 e。代码堆组 Δ 是只读的，故不会发生变化。事件 e 用于记录系统在一个指令周期内是否发生了陷阱执行操作，如果发生则记录下陷阱类型，否则为空。注意此处是在陷阱执行操作的周期记录事件，而不是在某条指令产生陷阱的周期内记录事件，在 SPARC 中，对陷阱的处理会在产生陷阱的指令周期的下一个周期执行。如某个周期内一条除法指令产生了除零错误，只是在 tt 寄存器内记录下陷阱类型，在下一个指令周期进行指令调度前进行陷阱处理，并记录下除零陷阱事件。多个单步执行的规则合成了多步执行的规则。图 3.9给出了多个指令周期的多步执行规则，其中多个单步执行产生的事件按序合为一个事件链表。事件链表就表示了执行多步后，系统执行过的陷阱的序列。

$$\boxed{\Delta \vdash S \models\!\xRightarrow{E}{}^{n} S'}$$

$$\frac{}{\Delta \vdash S \models\!\xRightarrow{\textbf{nil}}{}^{0} S}$$

$$\frac{\Delta \vdash S \models\!\xRightarrow{E}{}^{n} S' \qquad \Delta \vdash S' \Longrightarrow \textbf{abort}}{\Delta \vdash S \models\!\xRightarrow{E}{}^{n+1} \textbf{abort}}$$

$$\frac{\Delta \vdash S \Longrightarrow S'' \qquad \Delta \vdash S'' \models\!\xRightarrow{E}{}^{n} S'}{\Delta \vdash S \models\!\xRightarrow{\bot::E}{}^{n+1} S'}$$

$$\frac{\Delta \vdash S \xRightarrow{e} S'' \qquad \Delta \vdash S'' \models\!\xRightarrow{E}{}^{n} S'}{\Delta \vdash S \models\!\xRightarrow{e::E}{}^{n+1} S'}$$

图 3.9　多步执行

3.3.2 中断，陷阱执行和模式选择

3.3.2.1 中断

在指令周期初始会先处理从主板传来的中断请求，中断请求也属于陷阱的一种且优先级最低。由于执行陷阱时始终选择优先级最高的陷阱进行执行而且中断每次只能产生一个，因此如果机器中已经有陷阱发生，则一定不能产生中断。如果此时机器没有陷阱产生，则还需检查是否允许产生陷阱，以及中断的等级是否符合要求：

$$\mathsf{interrupt}(w,Q) \stackrel{def}{=\!=} \begin{cases} \mathsf{set_trap}(R\{tt \rightsquigarrow 16+w\}) & \textbf{if } \neg\mathsf{has_trap}(R), \mathsf{trap_enabled}(R), \\ & \quad 1 \leq w \leq 15, w = 15 \vee R(pil) < w \\ \bot & \textbf{otherwise} \end{cases}$$
$$\textbf{where } Q = (R,F)$$

interrupt 函数即为判断中断是否被允许的函数，输入参数 w 为中断请求的等级，如果中断请求满足条件，则在陷阱种类寄存器中记录下陷阱种类并将系统置为陷阱状态，如果中断请求不被满足，则返回空。如果中断请求被允许，即，由于中断请求也是陷阱的一种，与指令异常陷阱或用户触发的软件陷阱一样，交由陷阱执行函数进行处理。

3.3.2.2 陷阱执行

在中断请求被允许或上一指令周期内产生指令陷阱时，系统会进入陷阱状态，此时需要陷阱执行函数（exe_trap）进行处理：

$$\mathsf{exe_trap}(Q) \stackrel{def}{=\!=} \begin{cases} \textbf{let } (R',F') ::= \mathsf{right_win}(1,Q) \ \textbf{ in} & \\ \textbf{let } R'' ::= \mathsf{to_sup}(\mathsf{save_mode}(\mathsf{disable_trap}(R'))) \ \textbf{ in} & \\ (\mathsf{clear_trap}(\mathsf{save_pc_npc}(r_{17}, r_{18}, R'')), F') & \textbf{if } \mathsf{trap_enabled}(R) \\ \bot & \textbf{otherwise} \end{cases}$$
$$\textbf{where } Q = (R,F)$$

exe_trap 函数首先检查当前系统是否处于开中断状态，如果处于关闭中断状态则直接返回空值；否则首先将窗口右旋 1 次，而后关中断并将当前状态保存到 ps 寄存器中，进入管理者模式，最后将当前 pc、npc 寄存器的值保存到 r_{17}、r_{18} 寄存器中，并清除陷阱状态。保存 PC、nPC 的值时，还需要注意无效指令状态：

$$\mathsf{save_pc_npc}(r_m, r_n, R) \stackrel{def}{=\!=} \begin{cases} R\{r_m \rightsquigarrow R(pc)\}\{r_n \rightsquigarrow R(npc)\} & \textbf{if } \neg\mathsf{annuled}(R) \\ \mathsf{clear_annul}(R\{r_m \rightsquigarrow R(npc)\} & \\ \qquad \{r_n \rightsquigarrow R(npc+4)\}) & \textbf{otherwise} \end{cases}$$

如果当前系统处于无效指令状态，则保存在 r_{17}、r_{18} 中的值实际上不是 pc、npc 寄存器的值，而是 npc 寄存器的值和 npc 寄存器的值加 4，保存的同时需要清除无效指令状态。这就保证了如果一条指令 i 将系统置为了无效指令状态，同时还产生了陷阱，那么被跳过的指令不是接下来要执行的指令——陷阱处理函数中的首条指令，而是从陷阱中返回后执行的指令——代码堆中放置在 i 指令后的指令。exe_trap 函数执行前，会使用以下函数记录下陷阱种类作为事件标号：

$$\mathsf{get_tt}(Q) \stackrel{def}{=\!=\!=} R(tt) \ \ \textbf{where}\ Q = (R, F)$$

3.3.2.3 模式选择

SPARCv8 中分为用户模式和管理者模式，二者所使用的代码堆和内存不同，所以在执行指令调度前，首先需要根据当前的模式选择对应的代码堆和内存。选择完毕后，接下来的指令调度在单个内存和代码堆上执行。

3.3.2.4 单步执行规则

综上，单步执行的规则主要有 3 条：

1. 如果有中断请求并请求成立，则此时一定为陷阱状态，记录下陷阱种类并执行陷阱执行函数，而后进行指令调度：

$$\frac{\mathsf{interrupt}(w, Q) = Q' \quad \mathsf{get_tt}(Q') = w \quad \mathsf{exe_trap}(Q') = Q'' \quad C_s \vdash (M_s, Q'', D) \bullet\!\!\longrightarrow (M_s', Q''', D')}{(C_u, C_s) \vdash ((M_u, M_s), Q, D) \stackrel{w}{\Longrightarrow} ((M_u, M_s'), Q''', D')}$$

2. 如果系统为陷阱状态，记录下陷阱种类并则执行陷阱执行函数，而后进行指令调度：

$$\frac{\mathsf{has_trap}(Q) \quad \mathsf{get_tt}(Q) = w \quad \mathsf{exe_trap}(Q) = Q' \quad C_s \vdash (M_s, Q', D) \bullet\!\!\longrightarrow (M_s', Q'', D')}{(C_u, C_s) \vdash ((M_u, M_s), Q, D) \stackrel{w}{\Longrightarrow} ((M_u, M_s'), Q'', D')}$$

3. 如果中断请求不被允许，或者没有中断请求且系统不处于陷阱状态，则直接进行代码堆和内存的选择和指令调度。

$$\frac{\neg\mathsf{has_trap}(Q) \quad \mathsf{usr_mode}(Q) \quad C_u \vdash (M_u, Q, D) \bullet\!\!\longrightarrow (M_u', Q', D')}{(C_u, C_s) \vdash ((M_u, M_s), Q, D) \Longrightarrow ((M_u', M_s), Q', D')}$$

$$\frac{\neg\mathsf{has_trap}(Q) \quad \mathsf{sup_mode}(Q) \quad C_s \vdash (M_s, Q, D) \bullet\!\!\longrightarrow (M_s', Q', D')}{(C_u, C_s) \vdash ((M_u, M_s), Q, D) \Longrightarrow ((M_u, M_s'), Q', D')}$$

注意，在规则 1 和规则 2 中，之所以不需要进行代码堆和内存的选择，是因为 exe_trap 函数执行之后，当前的模式一定为管理者模式，所以选择的代码堆和内存一定是管理者模式所对应的。

除了这 3 条规则以外，还可能由于调用 exe_trap 函数返回空值或指令调度异常使系统进入中止状态。全部的规则如图 3.10所示。

$$\boxed{\Delta \vdash S \stackrel{e}{\Longrightarrow} S'}$$

$$\frac{\mathsf{interrupt}(w,Q)=Q' \qquad \mathsf{exe_trap}(Q')=\bot}{(C_u,C_s)\vdash((M_u,M_s),Q,D)\Longrightarrow \textbf{abort}}$$

$$\frac{\mathsf{interrupt}(w,Q)=Q' \qquad \mathsf{exe_trap}(Q')=Q'' \qquad C_s\vdash(M_s,Q'',D)\bullet\!\!\longrightarrow \textbf{abort}}{(C_u,C_s)\vdash((M_u,M_s),Q,D)\Longrightarrow \textbf{abort}}$$

$$\frac{\begin{array}{l}\mathsf{interrupt}(w,Q)=Q' \qquad \mathsf{get_tt}(Q')=w \qquad \mathsf{exe_trap}(Q')=Q''\\ C_s\vdash(M_s,Q'',D)\bullet\!\!\longrightarrow(M_s',Q''',D')\end{array}}{(C_u,C_s)\vdash((M_u,M_s),Q,D)\stackrel{w}{\Longrightarrow}((M_u,M_s'),Q''',D')}$$

$$\frac{\mathsf{has_trap}(Q) \qquad \mathsf{exe_trap}(Q)=\bot}{(C_u,C_s)\vdash((M_u,M_s),Q,D)\Longrightarrow \textbf{abort}}$$

$$\frac{\mathsf{has_trap}(Q) \qquad \mathsf{exe_trap}(Q)=Q' \qquad C_s\vdash(M,Q',D)\bullet\!\!\longrightarrow \textbf{abort}}{(C_u,C_s)\vdash((M_u,M_s),Q,D)\Longrightarrow \textbf{abort}}$$

$$\frac{\begin{array}{l}\mathsf{has_trap}(Q) \qquad \mathsf{get_tt}(Q)=w \qquad \mathsf{exe_trap}(Q)=Q'\\ C_s\vdash(M_s,Q',D)\bullet\!\!\longrightarrow(M_s',Q'',D')\end{array}}{(C_u,C_s)\vdash((M_u,M_s),Q,D)\stackrel{w}{\Longrightarrow}((M_u,M_s'),Q'',D')}$$

$$\frac{\neg\mathsf{has_trap}(Q) \qquad \mathsf{usr_mode}(Q) \qquad C_u\vdash(M_u,Q,D)\bullet\!\!\longrightarrow \textbf{abort}}{(C_u,C_s)\vdash((M_u,M_s),Q,D)\Longrightarrow \textbf{abort}}$$

$$\frac{\neg\mathsf{has_trap}(Q) \qquad \mathsf{usr_mode}(Q) \qquad C_u\vdash(M_u,Q,D)\bullet\!\!\longrightarrow(M_u',Q',D')}{(C_u,C_s)\vdash((M_u,M_s),Q,D)\Longrightarrow((M_u',M_s),Q',D')}$$

$$\frac{\neg\mathsf{has_trap}(Q) \qquad \mathsf{sup_mode}(Q) \qquad C_s\vdash(M_s,Q,D)\bullet\!\!\longrightarrow \textbf{abort}}{(C_u,C_s)\vdash((M_u,M_s),Q,D)\Longrightarrow \textbf{abort}}$$

$$\frac{\neg\mathsf{has_trap}(Q) \qquad \mathsf{sup_mode}(Q) \qquad C_s\vdash(M_s,Q,D)\bullet\!\!\longrightarrow(M_s',Q',D')}{(C_u,C_s)\vdash((M_u,M_s),Q,D)\Longrightarrow((M_u,M_s'),Q',D')}$$

图 3.10　中断，执行陷阱和模式选择

3.3.3 延迟状态转移和无效指令状态的转移

3.3.3.1 延迟状态转移

在指令调度之前，首先需要进行延迟寄存器链表的例行检查操作：

$$\mathsf{exe_delay}(Q,D) \stackrel{def}{=\!=} \begin{cases} \textbf{let } R' ::= \mathsf{dwrite_psr}(w,R) \textbf{ in} & \\ (\mathsf{set_win}(w_{<4:0>},(R',F)),D') & \textbf{if } D=(0,psr,w)::D' \\ \textbf{let } R' ::= \mathsf{dwrite_tbr}(w,R) \textbf{ in} & \\ ((R',F)),D') & \textbf{if } D=(0,tbr,w)::D' \\ \textbf{let } R' ::= \mathsf{dwrite_wim}(w,R) \textbf{ in} & \\ ((R',F)),D') & \textbf{if } D=(0,wim,w)::D' \\ ((R\{\varsigma \rightsquigarrow w\},F),D') & \textbf{if } D=(0,\varsigma,w)::D', \\ & \quad \varsigma \neq psr \textbf{ or } tbr \textbf{ or } wim \\ \textbf{let } (Q',D'') ::= \mathsf{exe_delay}(Q,D') \textbf{ in} & \\ (Q',(n-1,\varsigma,w)::D'') & \textbf{if } D=(n,\varsigma,w)::D', n>0 \\ (Q,D) & \textbf{otherwise} \end{cases}$$
$$\textbf{where } Q=(R,F)$$

exe_delay 遍历延迟寄存器链表，对于延迟周期为 0 的元素，从链表中取出并对相应寄存器进行写入。对于延迟周期不为 0 的元素，将延迟周期减 1。

对于 y 寄存器、asr 寄存器，寄存器的全段均需要写入；对于 wim 寄存器，只需要写和当前窗口有关的段，即 0 至 N-1；对于 tbr 寄存器，只需要写入 tba 段；对于 psr 寄存器，et 和 pil 段在指令执行时已经被立即写入，故不需要在此处写入：

$$\begin{aligned} \mathsf{dwrite_psr}(w,R) &\stackrel{def}{=\!=} R\{n \rightsquigarrow w_{<23>}\}\{z \rightsquigarrow w_{<22>}\}\{v \rightsquigarrow w_{<21>}\} \\ &\qquad \{c \rightsquigarrow w_{<20>}\}\{s \rightsquigarrow w_{<7>}\}\{ps \rightsquigarrow w_{<6>}\} \\ \mathsf{dwrite_tbr}(w,R) &\stackrel{def}{=\!=} R\{tba \rightsquigarrow w_{<31:12>}\} \\ \mathsf{dwrite_wim}(w,R) &\stackrel{def}{=\!=} R\{wim_{<(N-1):0>} \rightsquigarrow w_{<(N-1):0>}\} \end{aligned}$$

3.3.3.2 无效指令状态转移

延迟状态转移之后需要进行无效指令状态转移。检查当前系统是否处于无效指令状态，如果处于无效指令状态，则直接跳过一条指令并解除无效指令状态；否则从代码堆中取出一条指令并执行。

3.3.3.3 延迟状态转移和无效指令状态的转移的规则

综上，延迟状态转移和无效指令状态的转移的规则主要有 2 条：

1. 执行延迟寄存器链表的例行检查操作，系统不处于无效指令状态，从代码堆中取出一条指令并执行：

$$\frac{\begin{array}{c}\mathsf{exe_delay}(Q,D)=(Q',D') \qquad \neg\mathsf{annuled}(Q') \\ C(\mathsf{cursor}(Q'))=i \qquad (M,Q',D')\circ\!\xrightarrow{i}(M,Q'',D'')\end{array}}{C\vdash(M,Q,D)\bullet\!\longrightarrow(M,Q'',D'')}$$

其中

$$\mathsf{cursor}(Q)\stackrel{def}{=\!=}R(pc)\ \ \textbf{where}\ Q=(R,F)$$

2. 执行延迟寄存器链表的例行检查操作，系统不处于无效指令状态，直接跳过一条指令并清除无效指令状态：

$$\frac{\mathsf{exe_delay}(Q,D)=(Q',D') \qquad \mathsf{annuled}(Q') \qquad \mathsf{clear_annul}(Q')=Q''}{C\vdash(M,Q,D)\bullet\!\longrightarrow(M,(\mathsf{next}(Q''),D')}$$

其中

$$\mathsf{next}(Q)\stackrel{def}{=\!=}\mathsf{next}(R)\ \ \textbf{where}\ Q=(R,F)$$

除了这 2 条规则之外，如果无法从代码堆中取出指令，即所给出的地址在代码堆中找不到指令，或者指令执行异常，则会使机器中止。全部规则如图 3.11 所示。

$$\boxed{C\vdash(M,Q,D)\bullet\!\longrightarrow(M',Q',D')}$$

$$\frac{\mathsf{exe_delay}(Q,D)=(Q',D') \qquad \neg\mathsf{annuled}(Q') \qquad C(\mathsf{cursor}(Q'))=\bot}{C\vdash(M,Q,D)\bullet\!\longrightarrow\textbf{abort}}$$

$$\frac{\begin{array}{c}\mathsf{exe_delay}(Q,D)=(Q',D') \qquad \neg\mathsf{annuled}(Q') \\ C(\mathsf{cursor}(Q'))=i \qquad (M,Q,D)\circ\!\xrightarrow{i}\textbf{abort}\end{array}}{C\vdash(M,Q,D)\bullet\!\longrightarrow\textbf{abort}}$$

$$\frac{\begin{array}{c}\mathsf{exe_delay}(Q,D)=(Q',D') \qquad \neg\mathsf{annuled}(Q') \\ C(\mathsf{cursor}(Q'))=i \qquad (M,Q',D')\circ\!\xrightarrow{i}(M,Q'',D'')\end{array}}{C\vdash(M,Q,D)\bullet\!\longrightarrow(M,Q'',D'')}$$

$$\frac{\mathsf{exe_delay}(Q,D)=(Q',D') \qquad \mathsf{annuled}(Q') \qquad \mathsf{clear_annul}(Q')=Q''}{C\vdash(M,Q,D)\bullet\!\longrightarrow(M,(\mathsf{next}(Q''),D')}$$

图 3.11 延迟状态和无效指令状态的转移

$(M,R) \xrightarrow{i} (M',R')$

$$\frac{[\![\beta]\!]_R = w \qquad \mathsf{word_aligned}(w) \qquad [\![\eta]\!]_R = false}{(M,R) \xrightarrow{\mathbf{bicc}\ \eta\ \beta} (M, \mathsf{next}(R))}$$

$$\frac{[\![\beta]\!]_R = w \qquad \mathsf{word_aligned}(w) \qquad [\![\eta]\!]_R = true}{(M,R) \xrightarrow{\mathbf{bicc}\ \eta\ \beta} (M, \mathsf{djmp}(w,R))}$$

$$\frac{[\![\beta]\!]_R = w \qquad \mathsf{word_aligned}(w) \qquad [\![\eta]\!]_R = false}{(M,R) \xrightarrow{\mathbf{bicca}\ \eta\ \beta} (M, \mathsf{set_annul}(\mathsf{next}(R)))}$$

$$\frac{[\![\beta]\!]_R = w \qquad \mathsf{word_aligned}(w)}{(M,R) \xrightarrow{\mathbf{bicca}\ al\ \beta} (M, \mathsf{set_annul}(\mathsf{djmp}(w,R)))}$$

$$\frac{[\![\beta]\!]_R = w \qquad \mathsf{word_aligned}(w) \qquad \eta \neq al \qquad [\![\eta]\!]_R = true}{(M,R) \xrightarrow{\mathbf{bicca}\ \eta\ \beta} (M, \mathsf{djmp}(w,R))}$$

$$\frac{[\![\beta]\!]_R = w \qquad \mathsf{word_aligned}(w) \qquad \mathsf{save_pc}(r_d, R) = R'}{(M,R) \xrightarrow{\mathbf{jmpl}\ \beta\ r_d} (M, \mathsf{djmp}(w,R'))}$$

$$\frac{}{(M,R) \xrightarrow{\mathbf{nop}} (M, \mathsf{next}(R))}$$

图 3.12 简单汇编指令的操作语义

3.3.4 汇编指令的操作语义

图 3.12和图 3.13给出了较为简单的汇编指令。这些汇编指令只与内存和寄存器的状态有关，没有窗口旋转，延迟写等特性，所以与帧寄存器链表和延迟寄存器链表状态无关。具体含义如下:

- **bicc** 规则要求地址表达式 β 的值 w 是字对齐的，如果条件判断表达式 η 判断成立，则延迟转移到新的地址；如果判断失败，则不进行转移:

$$\frac{[\![\beta]\!]_R = w \qquad \mathsf{word_aligned}(w) \qquad [\![\eta]\!]_R = false}{(M,R) \xrightarrow{\mathbf{bicc}\ \eta\ \beta} (M, \mathsf{next}(R))}$$

$$\frac{[\![\beta]\!]_R = w \qquad \mathsf{word_aligned}(w) \qquad [\![\eta]\!]_R = true}{(M,R) \xrightarrow{\mathbf{bicc}\ \eta\ \beta} (M, \mathsf{djmp}(w,R))}$$

其中，word_aligned 表示传入参数是字对齐的，定义如下:

$$\mathsf{word_aligned}(w) \stackrel{def}{=\!=} w_{<1:0>} = 0$$

$\boxed{(M,R) \xrightarrow{i} (M',R')}$

$$\frac{[\![\beta]\!]_R = w \qquad \mathsf{word_aligned}(w) \qquad w \in dom(M) \qquad R' = R\{r_d \rightsquigarrow M(w)\}}{(M,R) \xrightarrow{\mathbf{ld}\ \beta\ r_d} (M, \mathsf{next}(R'))}$$

$$\frac{[\![\beta]\!]_R = w \qquad \mathsf{word_aligned}(w) \qquad w \in dom(M) \qquad M' = M\{w \rightsquigarrow [\![r_d]\!]_R\}}{(M,R) \xrightarrow{\mathbf{st}\ r_d\ \beta} (M', \mathsf{next}(R))}$$

$$\frac{[\![\gamma]\!]_R \neq \bot \qquad [\![\eta]\!]_R = false}{(M,R) \xrightarrow{\mathbf{ticc}\ \eta\ \gamma} (M, \mathsf{next}(R))}$$

$$\frac{[\![\gamma]\!]_R = w \qquad [\![\eta]\!]_R = true}{(M,R) \xrightarrow{\mathbf{ticc}\ \eta\ \gamma} (M, \mathsf{set_user_trap}(w_{<6:0>}, R))}$$

$$\frac{\mathsf{sup_mode}(R) \qquad R' = R\{r_d \rightsquigarrow R(\varsigma)\}}{(M,R) \xrightarrow{\mathbf{rd}\ \varsigma\ r_d} (M, \mathsf{next}(R'))}$$

$$\frac{\mathsf{usr_mode}(R) \qquad \varsigma = y \ \mathbf{or}\ asr_i \qquad R' = R\{r_d \rightsquigarrow R(\varsigma)\}}{(M,R) \xrightarrow{\mathbf{rd}\ \varsigma\ r_d} (M, \mathsf{next}(R'))}$$

$$\frac{[\![\alpha]\!]_R = w \qquad w \neq 0 \qquad R' = \mathsf{udivcc}(R, r_s, \alpha, r_d)}{(M,R) \xrightarrow{\mathbf{udivcc}\ r_s\ \alpha\ r_d} (M, \mathsf{next}(R'))}$$

图 3.13 简单汇编指令的操作语义（续）

- **bicca** 规则除了要求地址表达式 β 的值 w 是字对齐之外，还根据条件表达式 η 的种类和判定条件，选择是否进行转移和是否进入无效指令状态。规则有 3 条：

  1. 如果条件判断表达式 η 判定失败，则不进行转移，但进入无效指令状态。
  $$\frac{[\![\beta]\!]_R = w \qquad \mathsf{word_aligned}(w) \qquad [\![\eta]\!]_R = false}{(M,R) \xrightarrow{\mathbf{bicca}\ \eta\ \beta} (M, \mathsf{set_annul}(\mathsf{next}(R)))}$$
  2. 如果条件判断表达式 η 是恒成立（al），则进行转移并进入无效指令状态。
  $$\frac{[\![\beta]\!]_R = w \qquad \mathsf{word_aligned}(w)}{(M,R) \xrightarrow{\mathbf{bicca}\ al\ \beta} (M, \mathsf{set_annul}(\mathsf{djmp}(w, R)))}$$
  3. 如果条件判断表达式 η 不是恒成立（al），但判定成功，则进行转移，

但不进入无效指令状态。

$$\frac{[\![\beta]\!]_R = w \qquad \mathsf{word_aligned}(w) \qquad \eta \neq al \qquad [\![\eta]\!]_R = true}{(M, R) \xrightarrow{\textbf{bicca}\ \eta\ \beta} (M, \mathsf{djmp}(w, R))}$$

- **jmpl** 规则要求地址表达式 β 的值 w 是字对齐的，满足条件后将当前 PC 值保存到寄存器 r_d 并延迟转移到新的地址 w。

$$\frac{[\![\beta]\!]_R = w \qquad \mathsf{word_aligned}(w) \qquad \mathsf{save_pc}(r_d, R) = R'}{(M, R) \xrightarrow{\textbf{jmpl}\ \beta\ r_d} (M, \mathsf{djmp}(w, R'))}$$

其中，save_pc 为保存 PC 值的函数，定义如下：

$$\mathsf{save_pc}(r_i, R) \stackrel{def}{=\!=} R\{r_i \rightsquigarrow R(pc)\}$$

- **nop** 规则不进行任何操作。

$$\frac{}{(M, R) \xrightarrow{\textbf{nop}} (M, \mathsf{next}(R))}$$

- **ld** 规则要求地址表达式 β 的值 w 是字对齐的，并且 w 在内存的定义域中。满足条件后，将内存中 w 地址的数据读出，存入 r_d 寄存器中。

$$\frac{[\![\beta]\!]_R = w \qquad \mathsf{word_aligned}(w) \qquad w \in dom(M) \qquad R' = R\{r_d \rightsquigarrow M(w)\}}{(M, R) \xrightarrow{\textbf{ld}\ \beta\ r_d} (M, \mathsf{next}(R'))}$$

- **st** 规则要求地址表达式 β 的值 w 是字对齐的，并且 w 在内存的定义域中。满足条件后，将 r_d 寄存器中的数据读出，存入内存中 w 地址处。

$$\frac{[\![\beta]\!]_R = w \qquad \mathsf{word_aligned}(w) \qquad w \in dom(M) \qquad M' = M\{w \rightsquigarrow [\![r_d]\!]_R\}}{(M, R) \xrightarrow{\textbf{st}\ r_d\ \beta} (M', \mathsf{next}(R))}$$

- **ticc** 规则需要判定条件表达式 η 的值，如果值为假，则不进行任何操作；如果值为真，则将陷阱表达式 γ 的值 w 的后 7 位作为标识，传入设置用户陷阱函数中。

$$\frac{[\![\gamma]\!]_R \neq \bot \qquad [\![\eta]\!]_R = false}{(M, R) \xrightarrow{\textbf{ticc}\ \eta\ \gamma} (M, \mathsf{next}(R))}$$

$$\frac{[\![\gamma]\!]_R = w \qquad [\![\eta]\!]_R = true}{(M, R) \xrightarrow{\textbf{ticc}\ \eta\ \gamma} (M, \mathsf{set_user_trap}(w_{<6:0>}, R))}$$

其中，set_user_trap 为设置用户陷阱的函数，定义如下：

$$\mathsf{set_user_trap}(k, R) \stackrel{def}{=\!=} \mathsf{set_trap}(R\{tt \rightsquigarrow 128 + k\})$$

- **rd** 规则分用户模式和管理者模式 2 种情况，在用户模式中，**rd** 指令只能读区到 y 寄存器和 asr 寄存器。如果满足上述要求，则将标志寄存器 ς 中的值存入 r_d 寄存器中。

$$\frac{\mathsf{sup_mode}(R) \qquad R' = R\{r_d \rightsquigarrow R(\varsigma)\}}{(M,R) \xrightarrow{\mathbf{rd}\ \varsigma\ r_d} (M, \mathsf{next}(R'))}$$

$$\frac{\mathsf{usr_mode}(R) \qquad \varsigma = y \ \mathbf{or}\ asr_i \qquad R' = R\{r_d \rightsquigarrow R(\varsigma)\}}{(M,R) \xrightarrow{\mathbf{rd}\ \varsigma\ r_d} (M, \mathsf{next}(R'))}$$

- **udivcc** 规则即带标志位的除法运算，首先将 y 寄存器的值和 r_s 寄存器的值连接成 64 位寄存器作为被除数，r_d 寄存器的值作为除数，进行除法运算，注意此处要求除数不能为零。运算完毕后，再将运算结构的状态，即是否为负数（n）、是否为零（z）、是否溢出（v）、是否进位（c），写入相应的寄存器中。

$$\frac{[\![\alpha]\!]_R = w \qquad w \neq 0 \qquad R' = \mathsf{udivcc}(R, r_s, \alpha, r_d)}{(M,R) \xrightarrow{\mathbf{udivcc}\ r_s\ \alpha\ r_d} (M, \mathsf{next}(R'))}$$

其中

$$\mathsf{udivcc}(R, r_s, \alpha, r_d) \stackrel{def}{=\!=} \mathbf{let}\ w = (R(y), [\![r_s]\!]_R) \div [\![\alpha]\!]_R\ \mathbf{in}$$
$$\begin{cases} \mathsf{set_icc}((0,1,0,0), R\{r_d \rightsquigarrow 0\}) & \mathbf{if}\ w = 0 \\ \mathsf{set_icc}((w_{<31>},0,0,0), R\{r_d \rightsquigarrow w\}) & \mathbf{if}\ w \neq 0, \\ & \quad w_{<63,32>} = 0 \\ \mathsf{set_icc}((1,0,1,0), R\{r_d \rightsquigarrow 2^{32}-1\}) & \mathbf{otherwise} \end{cases}$$

$$\mathsf{set_icc}(k, R) \stackrel{def}{=\!=} R\{n \rightsquigarrow w_n\}\{z \rightsquigarrow w_z\}\{v \rightsquigarrow w_v\}\{c \rightsquigarrow w_c\}$$
$$\mathbf{where}\ k = (w_n, w_z, w_v, w_c)$$

除了上述指令另外，还有一些是涉及到窗口旋转和延迟写特性的复杂指令，如图 3.14所示。具体含义如下：

- **save** 规则对窗口进行右旋操作，并将操作数表达式 α 的值 a 与 r_s 寄存器的值相加，写入 r_d 寄存器中。

$$\frac{\mathsf{dec_win}(R,F) = (R',F') \qquad [\![\alpha]\!]_R = a \qquad R'' = R'\{r_d \rightsquigarrow [\![r_s]\!]_R + a\}}{(M,(R,F),D) \circ\!\!\xrightarrow{\mathbf{save}\ r_s\ \alpha\ r_d} (M, (\mathsf{next}(R''), F'), D)}$$

- **restore** 规则对窗口进行左旋操作，并将操作数表达式 α 的值 a 与 r_s 寄存器的值相加，写入 r_d 寄存器中。

$$\frac{\mathsf{inc_win}(R,F) = (R',F') \qquad [\![\alpha]\!]_R = a \qquad R'' = R'\{r_d \rightsquigarrow [\![r_s]\!]_R + a\}}{(M,(R,F),D) \circ\!\!\xrightarrow{\mathbf{restore}\ r_s\ \alpha\ r_d} (M, (\mathsf{next}(R''), F'), D)}$$

- **rett** 规则首先要求系统在关中断状态并且处于管理者模式，并且地址表达式 β 的值 w 是字对齐的。满足上述条件后，对窗口进行左旋操作，开中断并恢复模式。

$$\frac{\neg\mathsf{trap_enabled}(R) \quad \mathsf{sup_mode}(R) \quad [\![\beta]\!]_R = w \quad \mathsf{word_aligned}(w) \quad \mathsf{rett_f}(R,F) = (R',F')}{(M,(R,F),D) \circ\!\xrightarrow{\textbf{rett}\ \beta} (M,(\mathsf{djmp}(w,R'),F'),D)}$$

其中

$$\mathsf{rett_f}(Q) \stackrel{def}{=\!=} \textbf{let}\ (R',F') ::= \mathsf{inc_win}(Q)\ \textbf{in}\ (\mathsf{restore_mode}(\mathsf{enable_trap}(R')),F')$$

- **wr** 规则同样要求用户模式时，只能对 y 寄存器和 asr 寄存器进行写入操作。除此之外，还要求写入 cwp 的值在窗口总数的索引范围之内。满足上述条件时，将操作数表达式 α 的值 a 与 r_s 寄存器的值进行亦或操作，结果记为 w。再将初始的延迟周期 X、寄存器名称、需要延迟写入的值 w 组成 3 元组，放入延迟寄存器链表中。注意对 psr 寄存器进行延迟写入时，与中断请求有关的 ET 段和 PIL 段所对应的 et 寄存器和 pil 寄存器需要立即写入。

$$\frac{\mathsf{usr_mode}(R) \quad \varsigma = y\ \textbf{or}\ asr_i \quad [\![\alpha]\!]_R = a \quad [\![r_s]\!]_R\ \textbf{xor}\ a = w \quad D' = \mathsf{set_delay}(\varsigma,w,D)}{(M,(R,F),D) \circ\!\xrightarrow{\textbf{wr}\ r_d\ \alpha\ \varsigma} (M,(\mathsf{next}(R),F),D')}$$

$$\frac{\mathsf{sup_mode}(R) \quad \varsigma \neq psr \quad [\![\alpha]\!]_R = a \quad [\![r_s]\!]_R\ \textbf{xor}\ a = w \quad D' = \mathsf{set_delay}(\varsigma,w,D)}{(M,(R,F),D) \circ\!\xrightarrow{\textbf{wr}\ r_d\ \alpha\ \varsigma} (M,(\mathsf{next}(R),F),D')}$$

$$\frac{\mathsf{sup_mode}(R) \quad [\![\alpha]\!]_R = a \quad [\![r_s]\!]_R\ \textbf{xor}\ a = w \quad w_{<4:0>} < N \quad D' = \mathsf{set_delay}(psr,w,D) \quad R' = R\{et \rightsquigarrow w_{<5>}\}\{pil \rightsquigarrow w_{<11:8>}\}}{(M,(R,F),D) \circ\!\xrightarrow{\textbf{wr}\ r_s\ \alpha\ psr} (M,(\mathsf{next}(R'),F),D')}$$

其中，将三元组放入延迟寄存器链表中操作的 set_delay 函数定义为:

$$\mathsf{set_delay}(\varsigma,w,D) \stackrel{def}{=\!=} (X,\varsigma,w) :: D$$

3.3.5 指令异常处理

指令异常包括异常陷阱和中止。

3.3.5.1 异常陷阱

指令异常陷阱是指指令调度时，指令产生的除零、地址对齐、窗口溢出等异常。这些错误不会导致机器运行中止，而是会将产生异常的种类标号存入陷阱种类寄存器中，在下一个周期执行陷阱时进入到对应的陷阱处理函数中进行相应的处理：

$$
\begin{aligned}
&illegal_ins \stackrel{def}{=\!=} 2 \quad && mem_not_align \stackrel{def}{=\!=} 7 \quad && win_overflow \stackrel{def}{=\!=} 5 \\
&privileged_ins \stackrel{def}{=\!=} 3 \quad && div_by_zero \stackrel{def}{=\!=} 42 \quad && win_underflow \stackrel{def}{=\!=} 6
\end{aligned}
$$

$$
\mathsf{trap_type}(i,Q) \stackrel{def}{=\!=} \begin{cases}
privileged_ins & \textbf{if } i = \textbf{rd}\ \varsigma\ r_d\ \textbf{ or }\ \textbf{wr}\ r_d\ \alpha\ \varsigma \\
& \quad [\![\alpha]\!]_R \neq \bot, \mathsf{usr_mode}(R), \\
& \quad \varsigma = wim\ \textbf{or}\ tbr\ \textbf{or}\ psr \\
& \textbf{if } i = \textbf{rett}\ \beta \\
& \quad \mathsf{trap_enabled}(R), \mathsf{usr_mode}(R) \\
illegal_ins & \textbf{if } \ i = \textbf{wr}\ r_s\ \alpha\ psr \\
& \quad [\![\alpha]\!]_R = w, ([\![r_s]\!]_R\ \textbf{xor}\ w)_{<4:0>} \geq N \\
& \textbf{if } i = \textbf{rett}\ \beta \\
& \quad \mathsf{trap_enabled}(R), \mathsf{sup_mode}(R) \\
win_overflow & \textbf{if } i = \textbf{save}\ r_s\ \alpha\ r_d \\
& \quad [\![\alpha]\!]_R \neq \bot, \mathsf{dec_win}(Q) = \bot \\
win_underflow & \textbf{if } i = \textbf{restore}\ r_s\ \alpha\ r_d \\
& \quad [\![\alpha]\!]_R \neq \bot, \mathsf{inc_win}(Q) = \bot \\
mem_not_align & \textbf{if } i = \textbf{ld}\ \beta\ r_d\ \textbf{ or }\ \textbf{st}\ r_d\ \beta\ \textbf{ or }\ \textbf{jmpl}\ \beta\ r_d \\
& \quad \textbf{or }\ \textbf{bicc}\ \eta\ \beta \\
& \quad [\![\beta]\!]_R = w, \neg\mathsf{word_aligned}(w) \\
div_by_zero & \textbf{if } i = \textbf{udivcc}\ r_s\ \alpha\ r_d \\
& \quad [\![\alpha]\!]_R = 0 \\
\bot & \textbf{otherwise}
\end{cases}
$$
$$\textbf{where } Q = (R, F)$$

$$
\begin{aligned}
\mathsf{unexpected_trap}(i,Q) \stackrel{def}{=\!=}\ & \textbf{let } w = \mathsf{trap_type}(i,Q) \textbf{ in} \\
& \begin{cases} (\mathsf{set_trap}(R\{tt \rightsquigarrow w\}), F) & \textbf{if } w \neq \bot \\ \bot & \textbf{otherwise} \end{cases} \\
& \textbf{where } \ Q = (R, F)
\end{aligned}
$$

trap_type 函数根据当前状态和需要执行的指令判断是否会产生异常陷阱，如果产生则返回陷阱类型，否则返回空。具体为：

- **rd** 指令和 **wr** 指令如果试图在用户模式下访问 wim、tbr、psr 寄存器，则

会产生特权指令错误（$privileged_ins$），**rett** 指令执行时如果系统处于开中断状态和用户模式下，同样会产生特权指令错误。

- **rett** 指令执行时如果系统处于开中断状态和管理者模式下，会产生非法指令错误。**wr** 指令对 psr 进行写操作时，如果要写入 cwp 寄存器的值超出窗口总数的索引范围，也会产生非法指令错误（$illegal_ins$）。
- **save** 指令和 **restore** 指令执行时需要对窗口进行旋转操作，如果旋转函数返回空值，即当前窗口不可用，则会分别产生窗口上溢错误（$win_overflow$）和窗口下溢错误（$win_underflow$）。
- **ld** 指令、**st** 指令、**jmpl** 指令和 **bicc** 指令中，均要求地址表达式的值是字对齐的，否则产生内存未对齐错误（mem_not_align）。
- **udivcc** 指令要求除数不能为 0，否则产生除零错误（div_by_zero）。

unexpected_trap 函数首先调用 trap_type 函数判断指令是否发生陷阱，如果发生则将陷阱类型写入 tt 寄存器中并将系统置为陷阱状态；否则返回空。

3.3.5.2 中止

中止是指指令产生错误使得机器不能再继续运行，指令是否会产生中止的判定函数如下：

$$\mathsf{abort_ins}(i,Q,M) \overset{def}{=\!=} \begin{cases} true & \textbf{if } i = \textbf{ld}\ \beta\ r_d \ \textbf{ or } \ \textbf{st}\ r_d\ \beta \ \textbf{ or } \ \textbf{jmpl}\ \beta\ r_d \ \textbf{ or } \ \textbf{rett}\ \beta \\ & \quad [\![\beta]\!]_R = \bot \\ true & \textbf{if } i = \textbf{ld}\ \beta\ r_d \ \textbf{ or } \ \textbf{st}\ r_d\ \beta \\ & \quad [\![\beta]\!]_R = w, w \notin dom(M) \\ true & \textbf{if } i = \textbf{udivcc}\ r_s\ \alpha\ r_d \ \textbf{ or } \ \textbf{save}\ r_s\ \alpha\ r_d \ \textbf{ or } \\ & \qquad \textbf{restore}\ r_s\ \alpha\ r_d \ \textbf{ or } \ \textbf{wr}\ r_d\ \alpha\ \varsigma \\ & \quad [\![\alpha]\!]_R = \bot \\ true & \textbf{if } i = \textbf{ticc}\ \eta\ \gamma \\ & \quad [\![\gamma]\!]_R = \bot \\ true & \textbf{if } i = \textbf{rett}\ \beta \\ & \quad [\![\beta]\!]_R = w, \neg\mathsf{word_aligned}(w) \ \vee \ \mathsf{usr_mode}(R) \vee \\ & \quad \mathsf{inc_win}(Q) = \bot \\ false & \textbf{otherwise} \end{cases}$$
$$\textbf{where } Q = (R, F)$$

中止情况分为 3 类：

- α、β、γ 表达式求值不满足数值范围条件。
- **ld** 指令和 **st** 指令访问内存时，地址不在定义域内。
- **rett** 指令从陷阱中返回时，如果地址未对齐或处于用户模式，亦或是下一个窗口不可用，则直接进入中止状态。

指令执行的异常陷阱规则和中止规则如下：

$$\frac{\mathsf{unexpected_trap}(i, Q) = Q'}{(M, Q, D) \overset{i}{\circ\!\!\longrightarrow} (M, Q', D)}$$

$$\frac{\mathsf{abort_ins}(i, Q, M)}{(M, Q, D) \overset{i}{\circ\!\!\longrightarrow} \textbf{abort}}$$

3.4 本章小结

本章先介绍了 SPARCv8 指令集合对应的抽象语法，然后介绍了 SPARCv8 指令执行的机器状态，最后介绍了指令的操作语义，从而完成了对 SPARCv8 指令集的形式化建模。此模型覆盖了 SPARCv8 中的众多特性，如提高函数调用效率的窗又寄存器机制，使得函数跳转更为灵活的延迟跳转机制，可将应用程序代码与操作系统代码在物理层次分开的模式机制，以及特殊的模式切换方式——陷阱等等。在对窗口的建模中，利用帧寄存器链表刻画某一时刻不可操作的窗口寄存器放，从而将通用寄存器和剩余窗口寄存器在模型中分离，使得建模结构上更佳清晰，并方便了对汇编代码的证明。

$$\boxed{(M,Q,D) \circ\!\xrightarrow{i} (M',Q',D')}$$

$$\frac{(M,R) \xrightarrow{i} (M',R')}{(M,(R,F),D) \circ\!\xrightarrow{i} (M',(R',F),D)}$$

$$\frac{\textsf{dec_win}(R,F)=(R',F') \qquad [\![\alpha]\!]_R = a \qquad R''=R'\{r_d \rightsquigarrow [\![r_s]\!]_R + a\}}{(M,(R,F),D) \circ\!\xrightarrow{\mathbf{save}\ r_s\ \alpha\ r_d} (M,(\textsf{next}(R''),F'),D)}$$

$$\frac{\textsf{inc_win}(R,F)=(R',F') \qquad [\![\alpha]\!]_R = a \qquad R''=R'\{r_d \rightsquigarrow [\![r_s]\!]_R + a\}}{(M,(R,F),D) \circ\!\xrightarrow{\mathbf{restore}\ r_s\ \alpha\ r_d} (M,(\textsf{next}(R''),F'),D)}$$

$$\frac{\begin{array}{l}\neg\textsf{trap_enabled}(R) \qquad \textsf{sup_mode}(R) \qquad [\![\beta]\!]_R = w \\ \textsf{word_aligned}(w) \qquad \textsf{rett_f}(R,F)=(R',F')\end{array}}{(M,(R,F),D) \circ\!\xrightarrow{\mathbf{rett}\ \beta} (M,(\textsf{djmp}(w,R'),F'),D)}$$

$$\frac{\begin{array}{l}\textsf{usr_mode}(R) \qquad \varsigma = y \ \mathbf{or}\ asr_i \qquad [\![\alpha]\!]_R = a \\ [\![r_s]\!]_R \ \mathbf{xor}\ a = w \qquad D' = \textsf{set_delay}(\varsigma,w,D)\end{array}}{(M,(R,F),D) \circ\!\xrightarrow{\mathbf{wr}\ r_d\ \alpha\ \varsigma} (M,(\textsf{next}(R),F),D')}$$

$$\frac{\begin{array}{l}\textsf{sup_mode}(R) \qquad \varsigma \neq psr \qquad [\![\alpha]\!]_R = a \\ [\![r_s]\!]_R \ \mathbf{xor}\ a = w \qquad D' = \textsf{set_delay}(\varsigma,w,D)\end{array}}{(M,(R,F),D) \circ\!\xrightarrow{\mathbf{wr}\ r_d\ \alpha\ \varsigma} (M,(\textsf{next}(R),F),D')}$$

$$\frac{\begin{array}{l}\textsf{sup_mode}(R) \qquad [\![\alpha]\!]_R = a \qquad [\![r_s]\!]_R \ \mathbf{xor}\ a = w \qquad w_{<4:0>} < N \\ D' = \textsf{set_delay}(psr,w,D) \qquad R' = R\{et \rightsquigarrow w_{<5>}\}\{pil \rightsquigarrow w_{<11:8>}\}\end{array}}{(M,(R,F),D) \circ\!\xrightarrow{\mathbf{wr}\ r_s\ \alpha\ psr} (M,(\textsf{next}(R'),F),D')}$$

$$\frac{\textsf{unexpected_trap}(i,Q)=Q'}{(M,Q,D) \circ\!\xrightarrow{i} (M,Q',D)}$$

$$\frac{\textsf{abort_ins}(i,Q,M)}{(M,Q,D) \circ\!\xrightarrow{i} \mathbf{abort}}$$

图 3.14　复杂汇编指令的操作语义和异常处理

第 4 章　形式化模型的性质及证明

在本章中，主要给出了与 SPARCv8 汇编语言的形式化模型有关的一些性质及证明。这些证明均已经在 Coq 实现，在此处只给出定理及简要证明思路。第一节给出了确定性定理及定理的证明思路，确定性定理主要是说明操作语义的确定性，在之后对汇编语言的验证过程中起重要的作用。第二节给出与安全性有关的隔离性定理及定理的证明思路，隔离性定理刻画了 SPARCv8 中用户模式与管理者模式内存空间隔离的特性。这个特性杜绝了处于用户模式时，对管理者模式的内存进行读写的可能。

4.1　确定性

在系统执行过程中，如果没有中断请求到来，那么系统的执行所带来的状态转换是确定的。即如果两个系统初始状态相同，那么同时执行 n 步后，状态亦相同。虽然中断请求的到来是非确定性的，但中断请求如果被允许，则一定会由于执行陷阱而记录下相应的事件。如果我们要求两个系统执行过程中，事件序列相同，则两个系统对于非确定性的中断处理亦相同，依然具有运行 n 步后，状态相同的性质，此即确定性定理。确定性定理说明了如果系统执行若干步后产生的事件序列相同，则末状态一定相同：

定理 4.1 (确定性)　如果 $\Delta \vdash S \overset{E}{\Longmapsto}{}^{*} S_1$，$\Delta \vdash S \overset{E}{\Longmapsto}{}^{*} S_2$，则有 $S_1 = S_2$。

其中 $\Delta \vdash S \overset{E}{\Longmapsto}{}^{*} S'$ 被定义为：存在 n，使得 $\Delta \vdash S \overset{E}{\Longmapsto}{}^{n} S'$。

证明

首先展开 $\Delta \vdash S \overset{E}{\Longmapsto}{}^{*} S'$ 定义，对 n 进行归纳。

n = 0 时显然成立。

需证明 n = N 时成立时，n = N+1 时成立。展开多步执行规则的定义（见图 3.9），此时只需要证明在单步执行第 N+1 步时，满足确定性定理即可。

单步执行首先需证明两个初始状态均为 S 的机器，在执行中断请求（interrupt）后末状态相同，而后需证明执行陷阱执行函数（exe_trap）后末状态相同，再证明模式选择和执行指令调度后末状态相同。其中中断请求与陷阱执行只需展开定义便可证明。

由于两个机器初始状态相同，所以二者模式相同，故必然会选择相同的代码堆和内存。而后，我们需要证明指令调度前的延迟寄存器链表的例行检查操作 (exe_delay) 执行后末状态相同，以及系统处于无效指令状态 (annuled)，进行无效指令状态转移后末状态相同，这些证明只需展开定义即可。

最后证明指令调度后末状态相同。证明需要将需要进行调度的指令进行情况分析，列举出所有可能的指令，再分别进行证明。 □

4.2 隔离性

在 SPARCv8 中分为管理者模式和用户模式。我们可以使在 SPARCv8 机器上运行的操作系统处于管理者模式，而将在操作系统中运行的应用程序处于用户模式。由于管理者模式和用户模式使用的代码堆和内存均不同，故二者会运行不同的程序片段。除此之外，由于管理者模式权限比用户模式更大，可以拥有对 PSR、TBR 等寄存器的读写权限，出于安全因素考虑，运行在用户模式的机器转移到管理者模式时，其转移路径必须由管理者所感知，否则在操作系统运行中，会出现应用程序随意访问并操作系统内核代码和存储空间的可能，造成安全问题。

在 SPARCv8 中，用户模式转移到管理者模式的路径只有一个，即通过触发陷阱的方式，将控制权转移到管理者模式。我们先证明这个性质。即证明，在没有陷阱产生时，处于用户模式下运行的系统执行任意步，仍然处在用户模式下，永远不能到达管理者模式，从而也就不能访问管理者模式的空间。

首先给出用户模式运行 n 步所需条件的定义：

$$\begin{aligned}\Delta \vdash S \overset{}{\longleftrightarrow}{}^{n} S' \stackrel{def}{=\!=\!=} & \ \mathsf{usr_mode}(S) \wedge \mathsf{empty_DL}(S) \wedge \Delta \vdash S \overset{E}{\models\!\Longrightarrow}{}^{n} S' \\ & \wedge \mathsf{no_trap_event}(E)\end{aligned}$$

其中

$$\begin{aligned}\mathsf{empty_DL}(S) & \stackrel{def}{=\!=\!=} D = \mathbf{nil} \quad \mathbf{where}\ S = ((M_u, M_s), Q, D) \\ \mathsf{no_trap_event}(E) & \stackrel{def}{=\!=\!=} \forall e \in E, e = \bot\end{aligned}$$

在用户模式运行 n 步时，我们首先要求系统处于用户模式下。其次由于延迟写入的存在，我们会要求初始时延迟寄存器链表为空，否则如果其中有对 PSR 寄存器的修改，可能会使系统进入管理者模式。最后，我们要求系统运行若干步后，没有陷阱产生。

我们需要证明在上述条件的要求下，系统确实始始终于用户模式下：

定理 4.2 (用户模式运行) 如果 $\Delta \vdash S \Longleftrightarrow^n S'$，则 usr_mode$(S')$。

证明 证明思路与确定性定理证明思路一致，在此不再赘述。 □

由于 n 是对所有自然数成立，所以系统执行任意步后仍然处在用户模式下。故我们可以称 $\Delta \vdash S \Longleftrightarrow^n S'$ 为"系统处于用户模式运行 n 步"。这个状态在接下来说明模型的隔离性时会用到。

除此了证明用户模式切换管理者模式时的转换路径限制之外，我们还需要证明处于用户模式下运行的系统不会对管理者模式的数据有读权限和写权限，此即隔离性：

定理 4.3 (写隔离) 如果 $\Delta \vdash S \Longleftrightarrow^n S'$，则 sup_part_eq$(S, S')$。

定理 4.4 (读隔离) 如果 usr_code_eq(Δ_1, Δ_2)，usr_state_eq(S_1, S_2)，且 $\Delta_1 \vdash S_1 \Longleftrightarrow^n S_1'$，$\Delta_2 \vdash S_2 \Longleftrightarrow^n S_2'$，则 usr_state_eq$(S_1', S_2')$。

其中

$$
\begin{array}{lll}
\text{usr_state_eq}(S, S') & \overset{def}{=\!=} & Q = Q' \ \wedge \ M_u = M_u' \\
& & \textbf{where } S = ((M_u, M_s), Q, D) \ , \ S' = ((M_u', M_s'), Q', D') \\
\text{usr_code_eq}(\Delta, \Delta') & \overset{def}{=\!=} & C_u = C_u' \\
& & \textbf{where } \Delta = (C_u, C_s) \ , \ \Delta' = (C_u', C_s') \ , \\
\text{sup_part_eq}(S, S') & \overset{def}{=\!=} & M_s = M_s' \\
& & \textbf{where } S = ((M_u, M_s), Q, D) \ , \ S' = ((M_u', M_s'), Q', D')
\end{array}
$$

证明 证明思路与确定性定理证明思路一致，在此不再赘述。 □

写隔离定理要求处于用户模式下运行的系统运行任意步时，初始状态和系统运行任意多步后的状态两者管理者模式的内存数据相同，即在运行过程中不会对管理者模式的数据进行写操作。

读隔离定理要求两个系统状态 S_1、S_2 与用户模式运行有关的状态相同。处于用户模式运行任意步后末状态 S_1' 和 S_2' 与用户模式运行有关的状态亦相同。

4.3 本章小结

在本章中，主要给出了与 SPARCv8 汇编语言的形式化模型有关的一些性质及相应的证明。在第一节中，证明了该机器模型的语义行为在没有外部中断的情况下满足确定性，在第二节中，证明了用户模式和管理者模式下的代码执行满足隔离性。这些性质在一定程度上说明了此模型的正确性。

第 5 章　验证窗口溢出处理函数

在本章中，我们将对窗口溢出处理函数的 SPARCv8 汇编代码进行验证。在第一节中，我们将介绍窗口使用规则及窗口溢出的概念。在第二节中，我们将分别介绍窗口溢出具体处理方式，达到了怎样的预期效果。在第三节中，我们给出了窗口溢出处理函数的数学规范，解释为什么满足这样规范的代码能够解决窗口溢出的问题。第四节中，我们将使用这个数学规范，基于在第三章中给出的操作语义对窗口溢出处理函数进行验证。由于上溢处理和下溢处理具有一定的对称性和相似性，在本文中仅对上溢处理函数进行验证。

5.1　窗口溢出

在 SPARCv8 中，WIM 寄存器用来区分窗口是否被弃用。如果 WIM 寄存器中窗口标号所对应的位为 1，那么此窗口被弃用。在使用窗口时，如果将其中一个窗口弃用，窗口的 local 寄存器组不存储数据，环状窗口会被断开，剩余的窗口可以被看作一个类似于栈的结构。栈底为被弃用的窗口的下一个，栈指针指向当前窗口的上一个。当前窗口到栈底之间为已经使用的空间，其它部分为剩余空间。如图 5.1 所示。

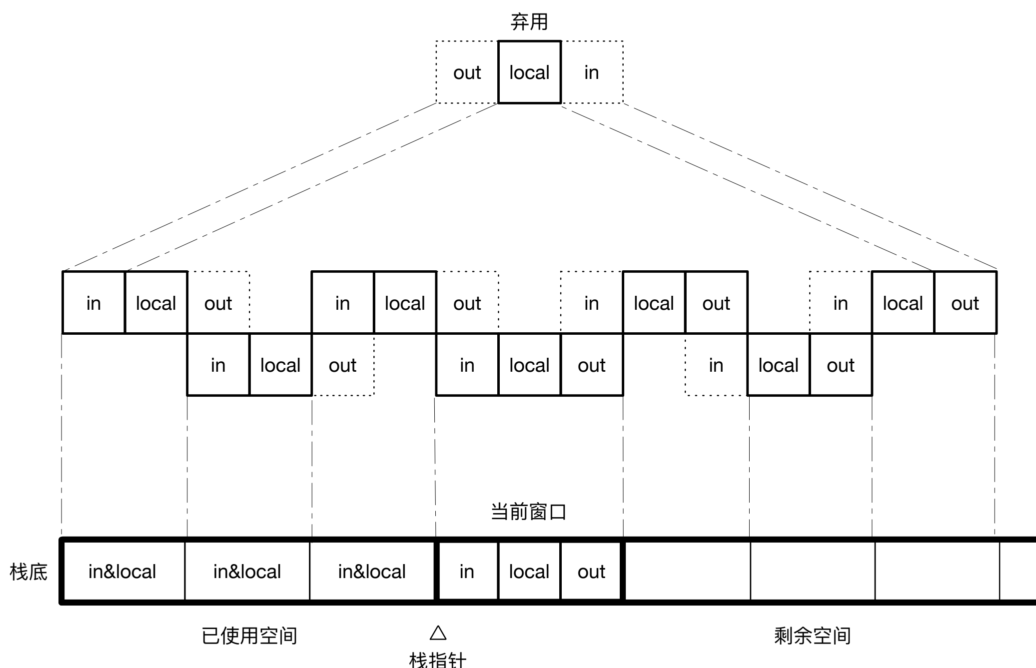


图 5.1　将窗口寄存器划分为栈空间和当前窗口

SAVE 指令需要保存上下文时，当前窗口的 in、local 寄存器组入栈，栈指针上移，out 寄存器组变为 in 寄存器组。栈剩余空间中，与 in 寄存器组相临的部分被划分为 local 和 out 寄存器组。RESTORE 指令需要恢复上下文时，栈指针下移，in、local 寄存器组出栈，当前窗口的 in 寄存器变为 out 寄存器组。如图 5.2 所示。

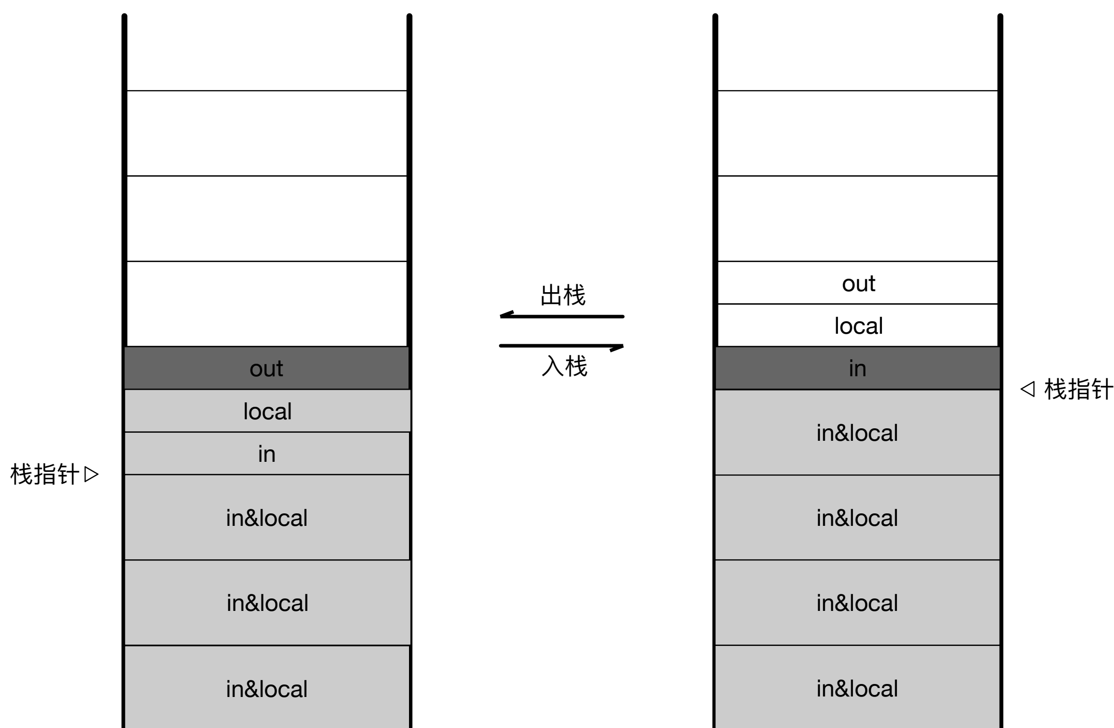


图 5.2　保存和恢复上下文时栈的变化

由于栈大小是有限的，最大存储数量为窗口数减 1，最小为 0，所以必然会有栈空和栈满的情况。当栈满时，如果再进行保存上下文的 SAVE 指令操作，则会触发窗口上溢陷阱；当栈空时，如果再进行恢复上下文的 RESTORE 指令操作，则会触发窗口下溢陷阱。上溢和下溢陷阱会调用相应的陷阱处理函数。

上溢陷阱处理函数需要将栈最底端的元素保存到内存中，为即将到来的 SAVE 操作腾出一个空间，能够继续保存上下文。下溢陷阱处理函数需要将上溢时保存到内存中的元素重新放回到栈中，使得 RESTORE 操作能够恢复相应的上下文。

5.2　窗口上溢的处理

窗口上溢的处理流程如下：

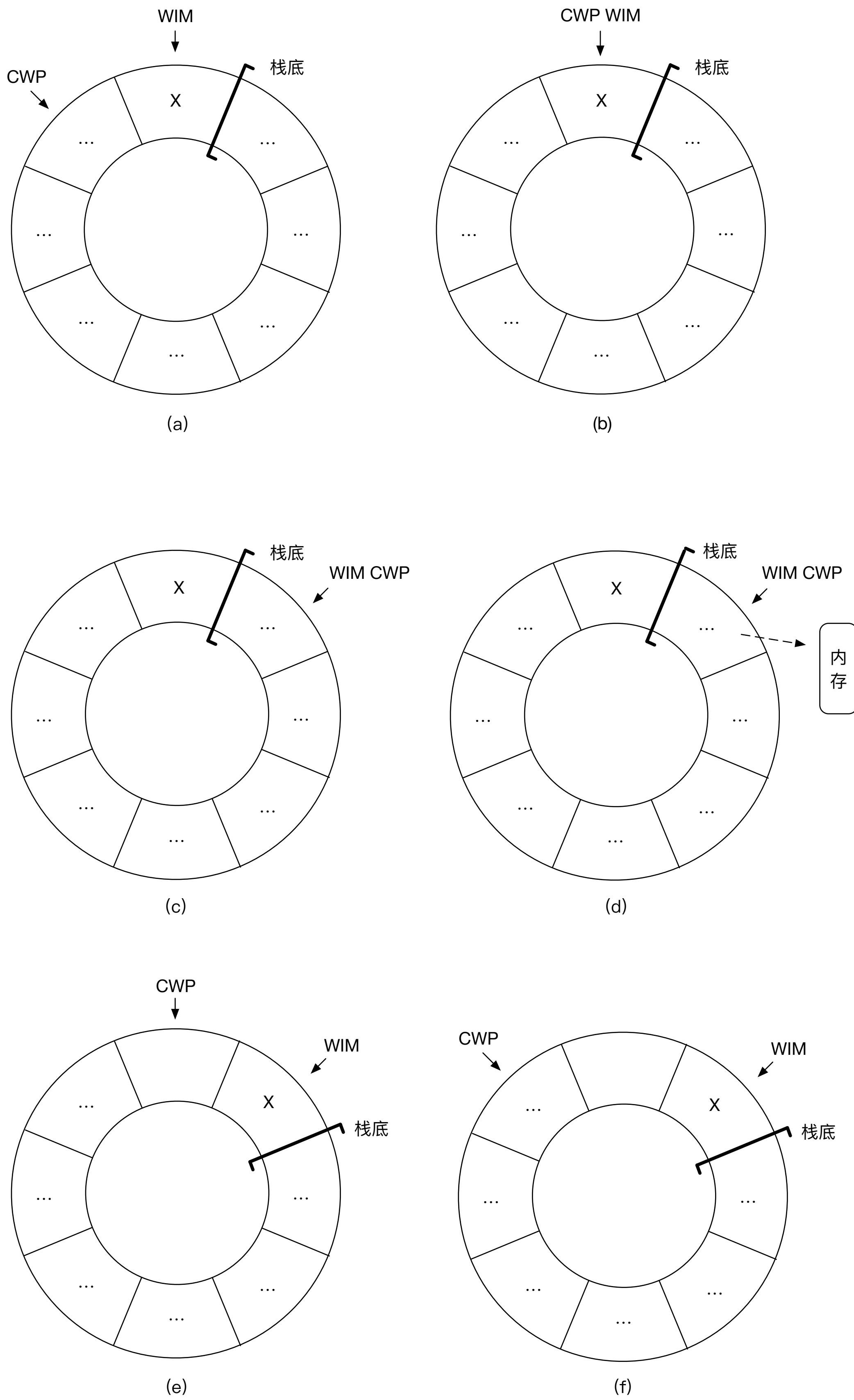


图 5.3 处理窗口上溢

- 图 5.3(a) 状态为执行 save 指令时，试图旋转到下一个窗口但下一个窗口不可用（wim 寄存器对应位值为 1），此时触发窗口上溢陷阱。
- 触发陷阱后，陷阱执行函数 exe_trap 会把窗口旋转到下一个，如图 5.3(b) 所示。此时进入窗口上溢的陷阱处理函数（代码 5.1）进行处理。

代码 5.1　窗口上溢出处理函数

```
1  windowoverflow:
2        mov        %wim,%l3
3        mov        %g1,%l7
4        srl        %l3,1,%g1
5        sll        %l3,OS_NWINDOWS-1,%l4
6        or         %l4,%g1,%g1
7        save
8        mov        %g1,%wim
9        nop
10       nop
11       nop
12       st         %l0,[%sp+0]
13       st         %l1,[%sp+4]
14       st         %l2,[%sp+8]
15       st         %l3,[%sp+12]
16       st         %l4,[%sp+16]
17       st         %l5,[%sp+20]
18       st         %l6,[%sp+24]
19       st         %l7,[%sp+28]
20       st         %i0,[%sp+32]
21       st         %i1,[%sp+36]
22       st         %i2,[%sp+40]
23       st         %i3,[%sp+44]
24       st         %i4,[%sp+48]
25       st         %i5,[%sp+52]
26       st         %i6,[%sp+56]
27       st         %i7,[%sp+60]
28       restore
29       mov        %l7,%g1
30       jmp        %l1
31       rett       %l2
```

- 代码 5.1中 2 ～ 6 行代码对 wim 寄存器的值进行循环右移一位操作，放入 $g1$ 寄存器中。5.1中 7 ～ 11 代码行先执行 save 指令，使得窗口指针指向下一个窗口，再将 $g1$ 寄存器中的值赋值给 wim 寄存器，将弃用窗口指向下一个窗口，完成后如图 5.3(c) 所示。注意此处的三个 nop 指令是为了等待延迟写入操作的完成。
- 代码 5.1中 12 ～ 27 行代码将栈底元素保存到内存中，栈底上移，如图 5.3(d) 所示。

- 代码 5.1中 28 ∼ 29 行代码执行 restore 指令，使得窗口指针指向上一个窗口，如图 5.3(e) 所示。
- 代码 5.1中 30 ∼ 31 行代码执行 rett 指令，使得窗口指针指向上一个窗口，如图 5.3(f) 所示。

至此，栈底元素保存到内存，腾出一个空间。当前窗口的下一个窗口变为可用状态，下下个窗口为弃用状态。

5.3 窗口上溢处理的数学规范

在本节中，我们将分别给出窗口上溢处理的数学规范，即前条件和后条件，用于下一节的正确性验证。

5.3.1 前条件

窗口上溢处理函数的前条件可以定义为：

$$\begin{aligned}\mathsf{overflow_pre_cond}(W) \overset{def}{=\!=} \; & \mathsf{single_mask}(R(cwp), R(wim)) \;\wedge\; \mathsf{handler_context}(R) \;\wedge \\ & \mathsf{normal_cursor}(R) \;\wedge\; \mathsf{set_function}(R(pc), \mathsf{windowoverflow}, C_s) \\ & \wedge\, \mathsf{align_context}(Q) \;\wedge\; D = \textbf{nil} \;\wedge\; \mathsf{length}(F) = 2N-3 \\ \textbf{where} \; & W = (\Delta, (\Phi, Q, D)),\; \Delta = (C_u, C_s),\; Q = (R, F)\end{aligned}$$

下面将对上述定义中的每个函数作具体说明。

5.3.1.1 唯一弃用窗口

定义唯一弃用窗口：

$$\mathsf{single_mask}(c, wim) \overset{def}{=\!=} 2^c = wim$$

single_mask 函数表示，当前系统只弃用了标号为 c 的窗口，其余窗口均可用。根据上一节的分析我们知道，进入窗口上溢处理函数之前需要执行 exe_trap 函数，执行 exe_trap 函数后，当前窗口指针指向弃用的窗口。故前条件之一为当前窗口弃用，其余窗口均可用，即 single_mask$(R(cwp), R(wim))$。

5.3.1.2 陷阱处理函数状态

在系统的运行过程中，要求当前窗口指针不能超出当前最大窗口范围。除此之外，所有陷阱处理函数都是在执行 exe_trap 函数执行后进入的，执行 exe_trap

函数后系统会有一些特有的状态，比如一定处于管理者模式下，一定是关陷阱状态等等，所以前条件的部分状态要求为：

$$\mathsf{handler_context}(R) \stackrel{def}{=\!=} 0 \leq cwp \leq 7 \land \mathsf{not_annuled}(R) \land \mathsf{trap_disabled}(R) \land \mathsf{no_trap}(R) \land \mathsf{sup_mode}(R)$$

5.3.1.3 程序入口

进入上溢陷阱处理函数之前，还要求 pc 寄存器所代表的程序指针指向陷阱处理函数的首行，并且 npc 寄存器与 pc 寄存器之间的值应该相差 4，即：

$$\mathsf{normal_cursor}(R) \land \mathsf{set_function}(R(pc), \mathsf{windowoverflow}, C_s)$$

其中

$$(Function) \quad j \quad ::= \quad \mathbf{nil} \mid i :: j$$

$$\mathsf{normal_cursor}(R) \stackrel{def}{=\!=} R(npc) = R(pc) + 4$$

$$\mathsf{set_function}(a, j, C) \stackrel{def}{=\!=} \begin{cases} C(a) = \mathbf{some}\ i \land \mathsf{set_function}(a+4, j', C) & \mathbf{if}\ j = i :: j' \\ \mathbf{true} & \mathbf{if}\ j = \mathbf{nil} \end{cases}$$

5.3.1.4 字对齐

由于 wr 指令、jmpl 指令、rett 指令所使用的地址表达式是字节对齐的，所以我们需要将其写入前条件中，其中 jmpl 指令，rett 指令分别需要使用 l_1 和 l_2 寄存器的值作为地址，wr 指令要求 sp 寄存器的值作为地址。在初始状态时，由于执行了一次 save 指令再执行 wr 指令，所以要求上一个窗口中的 sp 寄存器是字节对齐的。执行了一次 save 指令再执行一次 resotre 指令后，再执行 jmpl 指令和 rett 指令，所以要求当前窗口的 sp 寄存器是字节对齐的即可，即：

$$\begin{aligned} \mathsf{align_context}(Q) \stackrel{def}{=\!=}\ & \mathsf{word_aligned}(R(l1)) \land \mathsf{word_aligned}(R(l2)) \land \\ & \mathsf{word_aligned}(R'(sp)) \\ \mathbf{where}\ & Q = (R, F),\ (R', F') = \mathsf{right_win}(1, (R, F)) \end{aligned}$$

5.3.1.5 帧寄存器链表和延迟寄存器链表

在执行上溢处理函数前，我们还需要保证延迟寄存器链表链表为空。除此之外，当窗口为 N 个时，帧寄存器链表长度必定为 2N - 3，否则不符合窗口数量与窗口寄存器数量的对应关系。以上要求即：

$$\mathsf{align_context}(Q) \land D = \mathbf{nil} \land \mathsf{length}(F) = 2N - 3$$

在建模的语法和语义中均未对此做要求，故需要在这里进行限制。

5.3.2 后条件

根据上一节的分析我们知道，当上溢处理函数执行完毕后，下下个窗口为弃用状态，其余窗口均为可用状态。即：

$$\begin{aligned}\mathsf{overflow_post_cond}(W) &\overset{def}{=\!=} \mathsf{single_mask}(pre_cwp(2,R), R(wim)) \\ &\quad \textbf{where } W = (\Delta, (\Phi, Q, D)), Q = (R, F)\end{aligned}$$

只要满择这个条件，当窗口上溢处理函数执行完毕时，系统 pc 和 npc 寄存器的值分别为进入窗口上溢处理函数前 l_1 和 l_2 寄存器的值，也就是陷阱执行函数中将进入陷阱处理函数前的 pc、npc 值保存的位置。所以，执行窗口上溢处理函数后，会继续执行那条在之前出现上溢陷阱的 save 指令，如图 5.4所示。注意，之所以不在陷阱处理函数执行完毕后，直接把窗口指向下一个可用窗口，返回到 save 指令下一条位置，是因为在程序片段的 save 指令中还可能带有修改 sp 寄存器的操作，而我们在陷阱处理函数中无法知道它试图修改多少，所以需要返回到 save 指令执行位置，再执行一次。后条件告诉我们下个窗口为可用状态，所以再一次执行 save 指令必然不会触发窗口上溢陷阱。

图 5.4 窗口上溢陷阱处理过程

5.4 正确性验证

正确性验证需要证明，如果初始时系统状态满足前条件，则存在一个末状态，末状态中需要运行的函数已经运行完毕，并且末状态需要满足后条件。在验证窗口上溢函数时，前后条件在上一节中已经给出。窗口上溢函数运行完毕即系统在初始状态执行 30 步指令。故正确性验证需要证明：

定理 5.1 (上溢处理函数正确性)　如果有 overflow_pre_cond(Δ, S)，则存在 S'，E 使得 $\Delta \vdash S \overset{E}{\Longmapsto}{}^{30} S'$，并且满足 overflow_post_cond$(\Delta, S')$。

证明

系统执行 30 步指令需要分开一步步证明。

首先证明存在一个状态 S' 使得系统能执行一步，即存在性定理。由于系统处于关陷阱状态，如果有陷阱产生，则直接进入中止状态。所以在每一步中，需要证明当前执行的指令不会产生陷阱，以及不是中止指令 (abort_ins) 即可。

再证明每一步中满足一定条件。对于唯一弃用窗口:

- 初始时有 single_mask$(R(cwp), R(wim))$。
- 执行代码 5.1的 $2 \sim 11$ 行后，窗口移动到上一个窗口，废弃窗口也移动到上一个窗口，所以当前窗口仍然是唯一弃用的窗口，即 single_mask$(R(cwp), R(wim))$。
- 执行代码 5.1的 $12 \sim 28$ 行后，当前窗口移动到下一个窗口，所以移动过后，上一个窗口变为唯一弃用窗口，即 single_mask(pre_cwp$(1, R), R(wim)$)。
- 执行代码 5.1的 $29 \sim 31$ 行后，当前窗口移动到下一个窗口，所以移动过后，上上一个窗口变为唯一弃用窗口，即 single_mask(pre_cwp$(2, R), R(wim)$)，此即后条件。

所以系统执行完 30 步后，满足后条件，得证。

□

5.5　本章小结

在本章中，首先说明了在 SPARCv8 上运行的程序通常对窗口的使用方式，以及在该使用方式下窗口溢出的定义。再以相对应的窗口溢出处理函数为例，给出了代码片段的形式化规范 (前后条件)，基于操作语义证明了该代码片段满足给定前后条件。验证 SPARCv8 代码片段在一定程度上说明了建模的正确性和可用性。

第 6 章 Coq 实现

第三章，第四章，第五章中分别介绍了 SPARCv8 的形式化建模，模型性质的证明，窗口溢出处理函数的验证。这些工作均在 Coq 中进行了实现。其中形式化建模 947 行，性质证明 1611 行，验证窗口溢出处理函数 7114 行，总计 9672 行。代码的组织结构如图 6.1所示。本章主要是给出上述实现的部分 Coq 代码。第一小节中，给出了汇编指令语法的 Coq 代码。第二小节中，给出了汇编指令的操作语义的 Coq 代码。第三小节中，给出了证明确定性和隔离性的 Coq 代码。第四小节中，给出了验证窗口溢出处理函数的 Coq 代码。

文件名称	内容描述	行数
Asm.v	SPARCv8 的形式化建模	947
Props.v	确定性定理，用户模式运行定理，读隔离定理，写隔离定理，和存在型定理的证明	3033
Instance.v	SpaceOS2 中窗口溢出处理函数的验证	5761
Integers.v	整数库及相关定理（取自 CompCert）	4462
Maps.v	Map 库及相关定理（取自 CompCert）	204
Coqlib.v	Coq 基本库（取自 CompCert）	1615
LibTactics.v	Coq 中的一些常用策略（取自 Software Foundations [29]）	4797
int_auto.v	与整数库配套的一些自动化策略（取自 CertiμC/OS-II 验证项目）	673
math_sol.v	与整数计算相关的一些自动化策略（取自 CertiμC/OS-II 验证项目）	775
Makefile	工程编译文件	-
总计		22537

图 6.1　Coq 代码结构和代码量统计

6.1　汇编指令语法和记号

代码 6.1给出了对汇编指令语法建模的 Coq 代码。

代码 6.1 汇编指令语法建模的 Coq 实现

```
Inductive SparcIns: Type :=
  | bicc: TestCond -> AddrExp -> SparcIns
  | bicca: TestCond -> AddrExp -> SparcIns
  | jmpl: AddrExp -> GenReg -> SparcIns
  | ld: AddrExp -> GenReg -> SparcIns
  | st: GenReg -> AddrExp -> SparcIns
  | ticc: TestCond -> TrapExp -> SparcIns
  | save: GenReg -> OpExp -> GenReg -> SparcIns
  | restore: GenReg -> OpExp -> GenReg -> SparcIns
  | rett: AddrExp -> SparcIns
  | rd: Symbol -> GenReg -> SparcIns
  | wr: GenReg -> OpExp -> Symbol -> SparcIns
  | sll: GenReg -> OpExp -> GenReg -> SparcIns
  | srl: GenReg -> OpExp -> GenReg -> SparcIns
  | or: GenReg -> OpExp -> GenReg -> SparcIns
  | and: GenReg -> OpExp -> GenReg -> SparcIns
  | nop: SparcIns.
```

有了汇编指令语法，我们可以在 Coq 中写出具体的汇编指令。

Coq 中允许定义记号 (Notation)，使得 Coq 中表达式的语法在书写上更贴近于真实 SPARC 汇编指令语法。例如在定义了代码 6.2中的记号后，我们可以将代码 5.1的窗口上溢处理函数写为代码 6.3中的形式。

代码 6.2 关于表达式的部分记号

```
Notation " ri +ar rj " := (Aro ri (Or rj))
                         (at level 1) : asm_scope.
Notation " r +av n " := (Aro r (Ow n))
                         (at level 1) : asm_scope.
Notation " r 'ar' " := (Ao (Or r))
                         (at level 1) : asm_scope.
Notation " n 'av' " := (Ao (Ow n))
                         (at level 1) : asm_scope.
Notation " r 'r' " := (Or r)
                         (at level 1) : asm_scope.
Notation " n 'v' " := (Ow n)
                         (at level 1) : asm_scope
```

代码 6.3 窗口溢出处理函数的 Coq 代码

```
Definition overflow_handler := [
  rd wim l3;
  or g0 g1r l7;
  srl l3 ($1)v g1;
  sll l3 (Asm.N-i($1))v l4;
  or l4 g1r g1;
  save g0 g0r g0;
  wr g0 g1r wim;
  nop;
  nop;
  nop;
```

```
  st l0 sp_ar;
  st l1 sp+_av($4);
  st l2 sp+_av($8);
  st l3 sp+_av($12);
  st l4 sp+_av($16);
  st l5 sp+_av($20);
  st l6 sp+_av($24);
  st l7 sp+_av($28);
  st i0 sp+_av($32);
  st i1 sp+_av($36);
  st i2 sp+_av($40);
  st i3 sp+_av($44);
  st i4 sp+_av($48);
  st i5 sp+_av($52);
  st i6 sp+_av($56);
  st i7 sp+_av($60);
  restore g0 g0_r g0;
  or g0 l7_r g1;
  jmpl l1_ar g0;
  rett l2_ar
].
```

6.2 汇编指令的操作语义

代码 6.4给出了部分汇编指令操作语义的 Coq 代码。

代码 6.4 汇编指令操作语义的 Coq 实现

```

Inductive ArrowR :
            Memory * RegFile ->
            SparcIns ->
            Memory * RegFile -> Prop :=
  | Bicca:
      forall tc addr i w M R,
      i = bicca tc addr ->
      eval_AddrExp addr R = Some w ->
      word_aligned_R w ->
      tc <> al ->
      eval_TestCond tc R = true ->
      ArrowR (M,R) i (M,djmp w R)
  | Jmpl:
      forall ri addr i w M R R',
      i = jmpl addr ri ->
      eval_AddrExp addr R = Some w ->
      word_aligned_R w ->
      save_pc ri R = R' ->
      ArrowR (M,R) i (M,djmp w R')
  | Ld:
      forall ri addr i w v M R R',
      i = ld addr ri ->
      eval_AddrExp addr R = Some w ->
      word_aligned_R w ->
```

```
26        M w = Some v ->
27        R' = R#ri <- v ->
28        ArrowR (M,R) i (M,next R')
29   | Sll:
30        forall ri o rj i a w M R R',
31        i = sll ri o rj ->
32        eval_OpExp o R = Some a ->
33        (R#ri) <<i (get_range 0 4 a) = w ->
34        R' = R#rj <- w ->
35        ArrowR (M,R) i (M,next R')
36   | Nop:
37        forall i M R,
38        i = nop ->
39        ArrowR (M,R) i (M,next R)
40   | ... .
```

6.3 模型性质相关定理

代码 6.5给出了模型性质相关定理证明的部分 Coq 代码。

代码 6.5 定理证明的 Coq 代码

```
1
2  (* 多步执行的定义 *)
3  Inductive ArrowZ:
4               CodePair -> State -> EventList ->
5               nat -> State -> Prop :=
6   | Zero:
7        ...
8   | No_Event:
9        ...
10  | Has_Event:
11       ... .
12
13
14 (* 确定性定理 *)
15 Theorem Determinacy:
16    forall n E CP S S1 S2,
17    ArrowZ CP S E n S1 ->
18    ArrowZ CP S E n S2 ->
19    S1 = S2.
20 Proof.
21  ...
22 Qed.
23
24
25 (* 处于用户模式下运行n步的定义 *)
26 Definition ArrorWR:
27              CodePair -> State ->
28              nat -> State -> Prop:=
29    fun CP S n S' =>
30    exists E,usr_mode_S S /\ empty_DL S /\
31    ArrowZ CP S E n S' /\ no_trap E.
```

```


(* 用户模式运行定理 *)
Theorem UnderUserMode:
   forall CP S n S',
     ArrorWR CP S n S' ->
     usr_mode_S S'.
Proof.
  ...
Qed.


(* 写隔离定理 *)
Theorem NonExfiltration:
   forall CP S n S',
     ArrorWR CP S n S' ->
     sup_part_eq S S'.
Proof.
  ...
Qed.


(* 读隔离定理 *)
Theorem NonInfiltration:
  forall n CP1 CP2 S1 S1' S2 S2',
    usr_code_eq CP1 CP2 ->
    usr_state_eq S1 S2 ->
    ArrorWR CP1 S1 n S1' /\ ArrorWR CP2 S2 n S2' ->
    usr_state_eq S1' S2'.
Proof.
  ...
Qed.
```

6.4 验证窗口溢出处理函数

代码 6.6给出了窗口溢出处理函数验证的部分 Coq 代码。

代码 6.6 窗口溢出处理函数验证的 Coq 代码

```

Definition handler_context (R: RegFile) :=
  0 <= Int.unsigned (R#cwp) <= Int.unsigned(Asm.N)-1 /\
  not_annuled_R R /\ trap_disabled_R R /\
  no_trap_R R /\ sup_mode_R R.


Definition single_mask : Word -> Word -> Prop :=
  fun cwp wim =>
    ($1) <<i cwp = wim.


Definition align_context(O: RState) :=
  let (R,F) := O in
```

```
15     word_aligned_R R#l1 /\ word_aligned_R R#l2 /\
16   let (R',F') := right_win 1 (R,F) in
17     word_aligned_R R'#sp.
18
19
20 Definition normal_cursor(O: RState) :=
21   let (R,_) := O in R#npc = R#pc +i ($4).
22
23
24 Fixpoint set_function
25   (w: Address) (F: Function)(C: CodeHeap): Prop :=
26   match F with
27   | i::F' => C w = Some i /\
28            set_function (w +i ($4)) F' C
29   | nil => True
30   end.
31
32
33 (* 前条件的定义 *)
34 Definition overflow_pre_cond : Cond :=
35   fun W =>
36     let (CP,S) := W in
37     let (Cu,Cs) := CP in
38     let '(MP,Q,D) := S in
39     let (R,F) := Q in
40     set_function (cursor_Q Q) overflow_handler Cs /\
41     normal_cursor Q /\ handler_context R /\
42     align_context Q /\ single_mask R#cwp R#wim /\
43     D = nil /\ f_context F.
44
45
46 (* 后条件的定义 *)
47 Definition overflow_post_cond : Cond :=
48   fun W =>
49     let (CP,S) := W in
50     let (Cu,Cs) := CP in
51     let '(MP,Q,_) := S in
52     let (R,F) := Q in
53     single_mask (pre_cwp 2 R) R#wim.
54
55
56 (* 验证窗口上溢处理函数 *)
57 Theorem HandleOverflow:
58   forall CP S,
59     overflow_pre_cond (CP,S) ->
60     exists S' E ,Z__ CP S E 30 S'/\
61     overflow_post_cond (CP,S').
62 Proof.
63   …
64 Qed.
```

6.5 本章小结

在本章中给出了第三章中 SPARCv8 的形式化建模，第四章中模型性质的证明，和第五章中窗口溢出处理函数验证的部分 Coq 代码。同时还介绍了如何在 Coq 中使用记号，来使得在 Coq 中定义的语言的语法更接近于真实的语法。

第 7 章　总结

在本章中，主要给出了本文工作的总结以及未来工作的展望。其中，本文工作总结在第一小节中给出，进一步工作在第二小节中给出。

7.1　本文工作总结

嵌入式实时操作系统被广泛应用于各种嵌入式设备中，尤其是在航空航天等安全攸关领域。用形式化验证技术严格保证底层操作系统的正确性需要我们同时对 C 语言和内嵌汇编进行建模。

目前操作系统验证项目并没有验证汇编代码，只是在抽象状态上刻画了汇编代码的行为。如果要验证这部分汇编代码，需要对汇编指令的具体行为进行形式化建模。本文对 SPARCv8 指令集合进行形式化建模，使得我们能够在汇编层面上验证嵌入式实时操作系统（如 SpaceOS2）的正确性。

为了能保证嵌入式操作系统汇编代码和 C 源代码之间行为的一致性，我们还需要使用经过验证的可信编译器 CompCert 来编译保证。然而 CompCert 目前只支持 PowerPC、ARM 和 x86 汇编指令集作为其后端，并不支持 SPARCv8。本文对 SPARCv8 指令集合进行形式化建模，以便于将来 CompCert 后端对 SPARCv8 的扩展。

对 SPARCv8 的形式化建模中，正确性和可用性也十分关键。本文通过证明 SPARCv8 形式化模型中的确定性，隔离型等性质，在一定程度上说明了此模型的正确性。通过验证一个真实的操作系统的窗口溢出处理函数的代码片段，来说明此模型的可用性。

7.2　进一步工作

- **更多的指令**

  在本文中，目前只是将 SPARCv8 中部分指令进行了建模，剩余的指令包括整数运算指令，浮点运算指令，协处理器指令。出于工程项目的考虑，下一步希望能够将剩余部分代码的语法和操作语义给出。

- **完善证明框架**

  在本文中只验证了 SpaceOS2 的窗口上溢处理函数，共 31 行汇编代码，但

需要的 Coq 代码却达到了 7000 行之多，平均验证一行汇编代码需要 200 多行 Coq 代码。原因就是没有一个完善的证明框架，如断言，霍尔逻辑等等。下一步希望完善证明框架，更方便的验证汇编代码。

- **扩展 CompCert**

为了保证编译过程的正确性从而构建一个可信的操作系统，我们还需要保证源码和目标机器码之间的一致性。下一步希望可以将此模型接入 CompCert 后端，使得 CompCert 支持从 C 语言到 SPARCv8 汇编语言的编译。

参考文献

[1] ARM architecture. https://en.wikipedia.org/wiki/ARM_architecture.

[2] Crash-Proof Code. http://www2.technologyreview.com/news/423692/crash-proof-code/.

[3] LEON3. http://www.gaisler.com/index.php/products/processors/leon3.

[4] PowerPC. https://en.wikipedia.org/wiki/PowerPC.

[5] SpaceOS2. http://zhuanti.spacechina.com/n561816/n562345/n563069/c605321/content.html.

[6] SPARC. https://en.wikipedia.org/wiki/SPARC.

[7] Ssreflect. http://ssr.msr-inria.inria.fr.

[8] The CompCert formally-verified C compiler. https://github.com/AbsInt/CompCert.

[9] The Coq Proof Assistant. https://coq.inria.fr.

[10] The SPARC Architecture Manual Version 8. http://gaisler.com/doc/sparcv8.pdf.

[11] x86. https://en.wikipedia.org/wiki/X86.

[12] Reynald Affeldt and Naoki Kobayashi. Formalization and verification of a mail server in coq. In *Software Security—Theories and Systems*, pages 217–233. Springer, 2003.

[13] Robert Atkey. Coqjvm: An executable specification of the java virtual machine using dependent types. In *International Workshop on Types for Proofs and Programs*, pages 18–32. 2007.

[14] Véronique Benzaken, Évelyne Contejean, and Stefania Dumbrava. A coq formalization of the relational data model. In *European Symposium on Programming Languages and Systems*, pages 189–208. 2014.

[15] Hao Chen, Xiongnan Newman Wu, Zhong Shao, Joshua Lockerman, and Ronghui Gu. Toward compositional verification of interruptible os kernels and device drivers. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 431–447. 2016.

[16] Adam Chlipala. Modular development of certified program verifiers with a proof assistant. *ACM SIGPLAN Notices*, 41(9):160–171, 2006.

[17] Patryk Czarnik, Jacek Chrząszcz, and Aleksy Schubert. Cojaq: a hierarchical view on the java bytecode formalised in coq.

[18] Dhammika Elkaduwe, Gerwin Klein, and Kevin Elphinstone. Verified protection model of the

sel4 microkernel. In *Working Conference on Verified Software: Theories, Tools, and Experiments*, pages 99–114. 2008.

[19] Xinyu Feng and Zhong Shao. Modular verification of concurrent assembly code with dynamic thread creation and termination. *ACM SIGPLAN Notices*, 40(9):254–267, 2005.

[20] Xinyu Feng, Zhong Shao, Yuan Dong, and Yu Guo. Certifying low-level programs with hardware interrupts and preemptive threads. In *ACM SIGPLAN Notices*, volume 43, pages 170–182. 2008.

[21] Xinyu Feng, Zhong Shao, Alexander Vaynberg, Sen Xiang, and Zhaozhong Ni. Modular verification of assembly code with stack-based control abstractions. In *ACM SIGPLAN Notices*, volume 41, pages 401–414. 2006.

[22] Anthony Fox and Magnus O Myreen. A trustworthy monadic formalization of the armv7 instruction set architecture. In *International Conference on Interactive Theorem Proving*, pages 243–258. 2010.

[23] Zhe Hou, David Sanan, Alwen Tiu, Yang Liu, and Koh Chuen Hoa. An executable formalisation of the sparcv8 instruction set architecture: A case study for the leon3 processor. In *FM 2016: Formal Methods: 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings 21*, pages 388–405. 2016.

[24] Andrew Kennedy, Nick Benton, Jonas B Jensen, and Pierre-Evariste Dagand. Coq: the world's best macro assembler? In *Proceedings of the 15th Symposium on Principles and Practice of Declarative Programming*, pages 13–24. 2013.

[25] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220. 2009.

[26] Robert Jan Krebbers. *The C standard formalized in Coq*. Uitgever niet vastgesteld, 2015.

[27] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *ACM SIGPLAN Notices*, volume 41, pages 42–54. 2006.

[28] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[29] Benjamin C Pierce, Chris Casinghino, Michael Greenberg, Vilhelm Sjoberg, and Brent Yorgey. Software foundations. *Webpage:* `http://www.cis.upenn.edu/bcpierce/sf/current/index.html`.

[30] Xiaomu Shi. *Certification of an instruction set simulator*. PhD thesis, Université de Grenoble, 2013.

[31] Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W Appel. Compositional

compcert. *ACM SIGPLAN Notices*, 50(1):275–287, 2015.

[32] Fengwei Xu, Ming Fu, Xinyu Feng, Xiaoran Zhang, Hui Zhang, and Zhaohui Li. A practical verification framework for preemptive os kernels. In *International Conference on Computer Aided Verification*, pages 59–79. 2016.