Science of Computer Programming 187 (2020) 102371

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Formalizing SPARCv8 instruction set architecture in Coq [☆]

Jiawei Wang [a], Ming Fu [b], Lei Qiao [c], Xinyu Feng [d,*]

[a] *University of Science and Technology of China, Hefei, China*
[b] *Huawei Technologies Co., Ltd., Shanghai, China*
[c] *Beijing Institute of Control Engineering, Beijing, China*
[d] *Nanjing University, Nanjing, China*

A R T I C L E   I N F O

Article history:
Received 31 March 2018
Received in revised form 15 April 2019
Accepted 12 November 2019
Available online 19 November 2019

Keywords:
SPARCv8
Coq
Verification
Operational semantics

A B S T R A C T

The SPARCv8 instruction set architecture (ISA) has been widely used in various processors for workstations, embedded systems, and space missions. In order to formally verify the correctness of embedded operating systems running on SPARCv8 processors, one has to formalize the semantics of SPARCv8 ISA. In this article, we present our formalization of SPARCv8 ISA, which is faithful to the realistic design of SPARCv8. We also prove the determinacy and isolation properties with respect to the operational semantics of our formal model. In addition, we have verified that two trap handlers handling window overflow and window underflow satisfy the user's specifications based on our formal model. All of the formalization and proofs have been mechanized in Coq.

1. Introduction

Computer systems have been widely used in national defense, finance and other fields. Building high-confidence systems plays a significant role in the development of computer systems. Operating system kernel is the most foundational software of computer systems, and its reliability is the key in building high-confidence computer system.

In aerospace and other security areas, the underlying operating system is usually implemented in C and assembly languages. In existing OS verification projects, *e.g.*, CertiμC/OS-II [1] and seL4 [2], the assembly code is usually not modeled in order to simplify the formalization of the target machine. They use abstract specifications to describe the behavior of the assembly code to avoid exposing the details of underlying machines, *e.g.*, registers and stacks. Therefore, the assembly code in those kernels is not actually verified. To verify whether the assembly code satisfies its specifications, it is inevitable to formalize the semantics of the assembly instructions.

As a highly efficient and reliable microprocessor, the SPARCv8 [3] instruction set architecture has been widely used in various processors for workstations, embedded systems, and space missions. For instance, SpaceOS [4] running on SPARCv8 processors is an embedded operating system developed by Beijing Institute of Control Engineering (BICE) and deployed in the central computer of Chang'e-3 lunar exploration mission. On the one hand, to formally verify SpaceOS, we need to formalize the SPARCv8 instruction set and build the mathematical semantic model of the assembly instructions. On the other hand, to ensure the consistency between the behavior of the target assembly code and the C source code, we hope to use the certified compiler CompCert [5] to compile SpaceOS. However, CompCert does not support SPARCv8 at the backend.

[☆] This work is supported in part by grants from National Natural Science Foundation of China (NSFC) under Grant No. 61632005.
[*] Corresponding author.
E-mail address: xyfeng@nju.edu.cn (X. Feng).

https://doi.org/10.1016/j.scico.2019.102371
0167-6423/© 2019 Elsevier B.V. All rights reserved.

Extending CompCert to support SPARCv8 also requires us to formalize the SPARCv8 instruction set architecture. In this paper, we make the following contributions:

- We formalize the operational semantics of the SPARCv8 ISA. Our formal model is faithful to the behaviors of the instructions described in the SPARCv8 manual [3], including all the integer units with most of the features in SPARCv8, *e.g.*, windowed registers, delayed control transfer, and interrupts and traps.
- We prove that the operational semantics satisfies the determinacy property, and the execution in the user mode or supervisor mode satisfies the isolation property.
- We take the trap handlers for window overflow and window underflow as examples, and give their pre-condition and post-condition to specify the expected behaviors. Like proving programs with Hoare triples [6], we prove that these trap handlers satisfy the given pre-/post-conditions and do not throw any exceptions.
- All of the formalization and proofs have been mechanized in Coq [7]. They contain around 14000 lines of Coq scripts in total (measured by cloc [8]). The source code can be accessed via the link [9].

This article extends the conference paper in SETTA 2017 [10]. First, we give more details about our model, including the definitions of the windows rotation operation, delayed writes, trap and abort handling, *etc.*. Second, we present operational semantics rules for more SPARCv8 instructions, including some delayed transfer rules, abort rules, *etc.*. Third, we provide more detailed explanation about why the window overflow occurs, and how the window overflow trap handler handles it. Finally, to ease the proof burden, we give a tool called ‘coq2smt’ [11] that can translate lemmas about arithmetic propositions from Coq into SMT solvers such as Z3 [12], and use it to solve these lemmas. The tool contains around 6000 lines of Coq and Ocaml code in total (measured by cloc [8]). We use this tool to verify the window underflow trap handler, and it contains 5500 lines of Coq scripts.

Related work Fox and Myreen gave the ARMv7 ISA model [13]. They used monadic specification and formalized the instruction decoding and operational semantics. Narges Khakpour et al. proved some security properties of ARMv7 in the proof assistant tool HOL4, including the kernel security property, user mode isolation property, and so on. Andrew Kennedy et al. formalized the subset of x86 in Coq [14], and they used type classes, notations and the mathematics library Ssreflect [15]. The CompCert compiler also has the formal modeling of ARM and x86. There are lots of modeling work related to the x86 and ARM, but due to the specific features of SPARCv8, these x86 and ARM ISA models can not be used directly for the SPARCv8 ISA.

Zhe Hou et al. modeled the SPARCv8 ISA in the proof assistant Isabelle [16], which is close to our work. But their work is focused on the SPARCv8 processor itself, instead of the assembly code running on it. To verify the assembly code, we need a better definition on the syntax and the operational semantics. And the definition of machine state needs to be hierarchical and easy to use when we verify the code running on it. Additionally, they did not model the interrupt feature in SPARCv8, hence their model could not describe the non-determinism of the operational semantics caused by interrupts. Besides, our formalization of the SPARCv8 ISA is implemented in Coq, while CompCert is implemented in Coq too. We can use our Coq implementation to extend the CompCert at the backend to support SPARCv8 in the future.

There are several tools that integrate SMT solvers into Coq [17–19], but none of them are suitable for proving low-level code. To prove the low-level code, it is inevitable to deal with different bits of integers and various arithmetical operations. Base on [19], we develop a tool called ‘coq2smt’ that can translate dozens of lemmas for arithmetical operations on 8, 16, 32 and 64 bits integers in Coq to the SMT solver and solve it.

There are some other verification work at assembly level [20–22], which give the formal models of different subsets of x86 instructions and the behavior of the x86 interrupt management. They mainly study the verification techniques of assembly code, while the instruction set is relatively small. We formalized the SPARCv8 ISA by considering all the features of the integer unit of SPARCv8 [3]. In the next section, we give a brief overview of these features.

2. Overview of SPARCv8 ISA

The Scalable Processor Architecture (SPARC) is a reduced instruction set computing (RISC) instruction set architecture (ISA) originally developed by Sun Microsystems. It is widely used in the electronic systems of space devices for its high performance, high reliability and low power consumption. For example, LEON3 [23], a SPARCv8 architecture-based processor, developed by the European Space Research and Technology Center, is widely used in application-specific integrated circuits.

Compared to other architectures, SPARCv8 has the following unique mechanisms:

- A variety of control-transfer instructions (CTIs) and annulled delay instructions for more flexible function jumps.
- The register window and window rotation mechanism for swapping context more efficiently.
- Two modes, user mode and supervisor mode, for separating the application code and operating system code at the physical level.
- A variety of traps for swapping modes through a special trap table that contains the first 4 instructions of each trap handler.
- Delayed-write mechanism for delaying the execution of register write operation for several cycles.

These characteristics pose quite a few challenges for formal modeling. We use the example below to demonstrate the subtle control flow in SPARCv8.

Example The following function CALLER calls the function SUM3 to add three variables together.

```
CALLER:                        SUM3:
    ...
1   mov 1, %o0                 6   save %sp, -64, %sp
2   mov 2, %o1                 7   add %i0, %i1, %l7
3   call  SUM3                 8   add %l7, %i2, %l7
4   mov 3, %o2                 9   ret
5   mov %o0, %l7               10  restore %l7, 0, %o0
    ...
```

The function ELSCORRverbatim1ELSCORR requires three input parameters. When the ELSCORRverbatim0ELSCORR calls ELSCORRverbatim1ELSCORR, it places the first two arguments, then calls ELSCORRverbatim1ELSCORR (Line 3) before placing the third argument (Line 4). In other words, the call instruction will be executed before the mov instruction which places the last argument. The reason is that when we call an another function by using instructions such as call, it will record the address that is going to jump to in the current execution cycle. But the real transfer procedure will be executed in the next instruction cycle. In this case, the instruction call is placed in front of the instruction mov 3, but it's executed after the instruction mov 3. This feature is called "delayed transfer", which also happens at lines 9 and 10.

In the function SUM3, we use instructions save and restore (Lines 6 and 10) to save and restore the caller's context. When this program is running, both CALLER and SUM3 will have register windows as their contexts. Each of them has 8 in, 8 local and 8 out registers. And their windows are overlapping — the CALLER's out registers are the SUM3's in registers. When the CALLER needs to save the context and pass parameters to SUM3, it will put the parameters in the overlapping section, i.e. its out registers, and rotates the window so SUM3 will receive these arguments without copying the data. After rotating the window, the SUM3's register window is currently in use, and the non-overlapping portion of the CALLER's window is hidden from the programmer now. These steps above are implemented by the save instruction. By contrast, when the SUM3 needs to pass the return value to the CALLER, it will put the return value in the overlapping section and rotate the window too, and these steps are implemented by the restore instruction.

The semantics of the delayed transfer and the window rotation mechanism are quite tricky in SPARCv8. In addition, other special mechanisms of SPARCv8 mentioned above are complicated and their behaviors are non-trivial. Therefore, it is necessary to give a formal model of the SPARCv8 ISA, which is the basis of verifying the SPARCv8 code.

3. Modeling SPARCv8 ISA

The SPARCv8 instruction set provides programmers with an assembly programming language. In the SPARCv8 ISA, each instruction cycle consists of the following phases: First, the processor checks for interrupt requests and exception traps. Then, if a delayed-write instruction (an instruction that requires a certain period of delay before it writes to the corresponding register) arrives, the processor will execute it. And if there is an annulling control transfer (a transfer that requires one cycle delay to execute the control transfer), the processor will jump to the corresponding address. Finally, the processor reads an instruction given by the program counter from the memory that corresponds to the current mode and dispatches it. These instructions are generally divided into two categories, one contains the arithmetic instructions, such as the load/store instructions and the add instructions. These instructions account for a large proportion of the instruction set, but they only access the general-purpose register and the memory. The other category contains the instructions that access or write the processor state register which contains various fields that hold the status information, and the trap base register which contains the address to transfer to when a trap occurs.

3.1. Hierarchical modeling

In order to formalize the above characteristics, we use a hierarchical modeling approach. We take each phase of the instruction cycle described above as a layer of our model. In addition, we also divide the phase of instruction dispatch into two layers according to the above classification of instructions.

As shown in Fig. 1, from the top to the bottom, we first define the operational semantics of simple instructions which only access the register file and memory using the transition $(M, R) \xrightarrow{i} (M', R')$, where M stands for the memory and R stands for the register file and will be introduced in detail in Sec. 3.3.1. Here, the instructions i can only access the general-purpose registers, transfer registers and the memory, regardless of whether the memory belongs to which mode or whether the general-purpose register belongs to which window, etc..

Secondly, we lift the first layer and give the operational semantic of the specific instructions about the window registers and delayed writes using the transition $(M, Q, D) \circ\!\!\xrightarrow{i} (M', Q', D')$. Here Q is a tuple of the current register file and the other window registers, and D is used to store all the delayed write instructions in the current processor state. The window

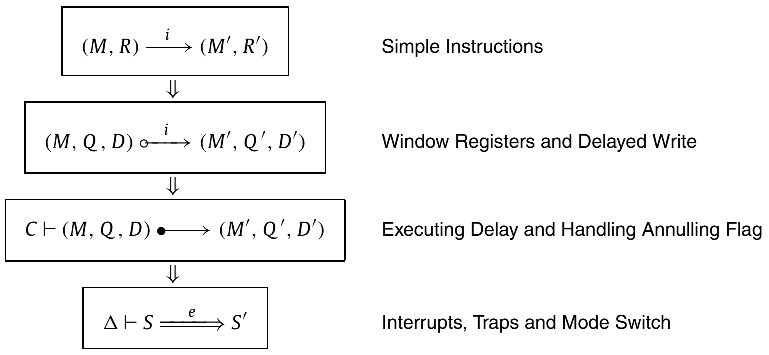
$(M, R) \xrightarrow{i} (M', R')$
Simple Instructions
$(M, Q, D) \overset{i}{\multimap} (M', Q', D')$
Window Registers and Delayed Write
$C \vdash (M, Q, D) \bullet\!\!\longrightarrow (M', Q', D')$
Executing Delay and Handling Annulling Flag
$\Delta \vdash S \xRightarrow{e} S'$
Interrupts, Traps and Mode Switch

Fig. 1. The structure of operational semantics.

$$
\begin{array}{rcll}
(\mathit{SparcIns}) & i & ::= & \textbf{ticc}\ \eta\ \gamma \mid \textbf{rett}\ \beta \mid \textbf{save}\ r_s\ \alpha\ r_d \mid \textbf{restore}\ r_s\ \alpha\ r_d \mid \textbf{ld}\ \beta\ r_d \mid \textbf{st}\ r_d\ \beta \\
 & & & \mid \textbf{rd}\ \varsigma\ r_d \mid \textbf{wr}\ r_d\ \alpha\ \varsigma \mid \textbf{bicc}\ \eta\ \beta \mid \textbf{bicca}\ \eta\ \beta \mid \textbf{jmpl}\ \beta\ r_d\ \mid \textbf{nop} \mid \ldots
\end{array}
$$

$$
\begin{array}{rcll@{\qquad}rcll}
(\mathit{GenReg}) & r & ::= & r_0 \mid \ldots \mid r_{31} & (\mathit{OpExp}) & \alpha & ::= & r \mid w \\
(\mathit{Symbol}) & \varsigma & ::= & psr \mid wim \mid tbr \mid y \mid asr & (\mathit{AddrExp}) & \beta & ::= & \alpha \mid r + \alpha \\
(\mathit{AsReg}) & asr & ::= & asr_0 \mid \ldots \mid asr_{31} & (\mathit{TrapExp}) & \gamma & ::= & r \mid r + r \mid r + w \mid w \\
(\mathit{Word}) & w & \in & \mathit{Int}32 & (\mathit{TestCond}) & \eta & ::= & al \mid eq \mid nv \mid ne \mid \ldots
\end{array}
$$

Fig. 2. The syntax of the SPARCv8 assembly language.

rotation instruction treats each window's 32 general-purpose registers as a whole register group, and performs the corresponding rotation operation without the knowledge of modes, interrupts, and so on. Similarly, when dispatching a delayed write instruction or a trap instruction, we only need to consider the registers involved in each instruction. The first and the second layer correspond to the third phase of the instruction cycle mentioned before, namely the instruction dispatch phase, where we dispatch all instructions. When these instructions are executed, some traps, delayed write commands or delayed transfers commands are generated. Instead of executing these commands at this stage, they will be executed at the beginning of the next instruction cycle, which are the third and fourth layers in our model.

Thirdly, we use the transition $C \vdash (M, Q, D) \bullet\!\!\longrightarrow (M', Q', D')$ to define the delayed execution and the handling of the annulling flag. Here C represents the code heap in the current mode. This layer corresponds to the second phase of the instruction cycle mentioned before.

Finally, we give the operational semantic rules of the interrupt, trap execution and mode switch as the transition $\Delta \vdash S \xRightarrow{-} S'$, which defines the whole behavior of the entire program. The state S contains three parts — a pair of memory, the register state, and the delay list. Δ stands for a pair of code heaps. This layer corresponds to the first phase of the instruction cycle mentioned before.

This hierarchical model is suitable for our verification work. For example, when we verify some instructions such as **ld**, **add**, *etc.*, we will only consider the register file and memory. If we put the exposed window register and the hidden window register on the same layer as [3] or [16] does, all the registers will always show up in the verification process.

In the following sub-sections, we first present the abstract syntax of SPARCv8 code. Then we define the machine state. Finally, we give the operational semantics rules for the instructions.

3.2. Syntax

Fig. 2 shows the syntax of the SPARCv8 assembly language. Here we only give some typical instructions i that show the key features introduced in Sec. 2. Others can be found in the Coq implementations [9]. **ticc** triggers a software trap, and **rett** returns from a trap. These two instructions are often used to invoke a system call and return from it. **save** (or **restore**) saves (or restores) the caller's context by rotating the register window. **ld** (or **st**) loads (or stores) values from (or to) the memory. **rd** (or **wr**) reads (or writes) some specific registers, which are defined as $Symbol$. The write by **wr** may be delayed for several cycles, as explained in Sec. 3.3.2. **bicc** makes a delayed control transfer if the condition holds, otherwise it executes the following code. **bicca** is similar to **bicc**, but it may annul the next instruction in the following code under some conditions. **jmpl** saves the current location and jumps to the destination. **nop** does nothing.

The register names in the instruction consist of two parts - general registers and symbol registers. r stands for general registers (*GenReg*), with names ranging from r_0 to r_{31}. These registers also have some aliases, as shown in Table 1. Note that the r_0 (g_0) register is a special register, Its value is always 0. So the read operation on this register always returns 0, and the write operation does noting, as shown below.

Table 1
General registers and corresponding aliases.

General register	r_0 - r_7	r_8 - r_{15}	r_{16} - r_{23}	r_{24} - r_{31}	r_{14}	r_{30}
Alias	g_0 - g_7	o_0 - o_7	l_0 - l_7	i_0 - i_7	sp	fp

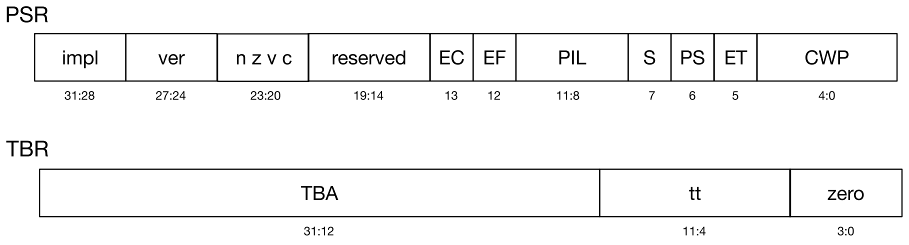

Fig. 3. PSR and TBR fields.

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

ς represents symbol registers (*Symbol*), it contains the processor state register (*psr*), window invalid mask register (*wim*), trap base register (*tbr*), multiply/divide register (y) and ancillary state registers (*asr*). *asr* represents 32 ancillary registers (*AsReg*) which are used to store the processor's ancillary state, with names ranging from asr_0 to asr_{31}. wim is a 32-bit register, and each bit indicates whether a window that corresponding to its index is valid. A register window with label w is *invalid* if the w bit of register wim is 1. *psr* and *tbr* represent the processor state register and the trap base register respectively. These two registers are also 32 bits, and they contain several fields to represent the different status of the processor, as shown in Fig. 3. The PIL field identifies the interrupt level above which the processor will accept an interrupt; the S field determines whether the processor is in supervisor or user mode; the PS field stores the previous mode of the processor; The ET field determines whether traps are enabled; The CWP field identifies the current window. The TBA field represents the trap base address, which is established by supervisor software; the tt field records the trap type if a trap occurred.

The expressions in these instructions contain four parts - conditional expressions, address expressions, operand expressions and trap expressions. The conditional expression η can be always (*al*), never (*ne*), equal (*eq*), not equal (*ne*), *etc.*. To evaluate the conditional expression, we need to determine whether the given condition is hold by reading the conditional registers (n, z, v, c) in *psr*, as shown below.

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

The address expressions, operand expressions and trap expressions in these instructions are defined as *AddrExp*, *OpExp* and *TrapExp*. To simplify the syntax of our model, we do not restrict the range of the immediate numbers in these expressions in syntax, but in the procedure of expression evaluation, as shown below. When these expressions are evaluated, if the immediate number w is out of range, the function will return $\perp$. Here w stands for 32-bit integers (*Word*).

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

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

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

Note that the `call`, `mov`, and `ret` instructions in the example in Sec. 2 are not given in the syntax, since they are all synthetic instructions, which can be defined from the basic instructions [3].

3.3. Machine states

As shown below, the whole world W contains two parts, namely the state S and a pair of code heap Δ, to represent the changeable parts and unchangeable parts of the system, respectively. The state S contains three parts - a pair of memory Φ, the register state Q, and the delay list D.

$$\begin{array}{rrcl} (World) & W & ::= & (\Delta, S) \\ (State) & S & ::= & (\Phi, Q, D) \\ (CodePair) & \Delta & ::= & (C_u, C_s) \\ (MemPair) & \Phi & ::= & (M_u, M_s) \end{array} \qquad \begin{array}{rrcl} (CodeHeap) & C & \in & Label \rightharpoonup SparcIns \\ (Memory) & M & \in & Address \rightharpoonup Word \\ (Label) & l & \in & Word \\ (Address) & a & \in & Word \end{array}$$

Since there are two modes in SPARCv8, namely user mode and supervisor mode, the full memory and code heap are split into two parts for these two modes respectively. C represents the code heap, which maps the labels to the instructions. M represents the memory, which maps the addresses to words. C represents the code heap, which maps the labels to the instructions. Labels and addresses are all 32-bit integers.

Next, we will introduce the register state Q and the delay list D.

3.3.1. Register state

The register state contains two parts, namely the register file R and the frame list F, as shown below.

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

As we mentioned in the example in Sec. 2, when we use instruction **save** to switch from the caller's context to the callee's context, the system will expose the caller's window and hide the callee's window that not in the overlapping section instead of copying data between them. So, we use register file to represent these registers that can be accessed now, and use frame list to represent the rest.

Register file As shown below, a register file R is modeled as a total function mapping register names to 32-bit integers. Register name (q) contains the general registers and symbol registers, which are explained before. It also contains the program counter pc, the next program counter npc, the trap flag τ and annulling flag κ.

$$(RegFile) \quad R \in RegName \rightarrow Word \qquad (RegName) \quad q \quad ::= \quad r \mid \varsigma \mid pc \mid npc \mid \kappa \mid \tau$$

SPARCv8 uses two program counters, *viz.*, pc and npc to control the execution. pc contains the address of the instruction currently being executed, while npc holds the address of the next instruction (assuming a trap does not occur). According to the type of the current running instruction, we have three different updates shown as below for the stepping forward of pc and npc.

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

The function next defines the change of program counters when no transfer occurs. It updates pc with npc and increases npc by 4. If transfer occurs during the instruction execution, for example, if the evaluation of conditional expression returns **true** when the system executes the instruction **bicc**, the function djmp will be executed. djmp updates pc with npc and sets npc to the target address. As mentioned in the example in Sec. 2, when we call a function, the target address w is stored in npc in the current execution cycle. Because the next instruction is fetched from pc, the transfer is not made immediately and is delayed to the next cycle instead. The *delayed transfer* is applied for all transfer instructions in SPARCv8. For jumping caused by traps, it will jump to the address of the corresponding trap handler using the function tbr_jmp, which sets pc to the entry address of the handler (tbr) and updates npc to "tbr + 4".

If an instruction throws exceptions during the execution, it usually causes a trap. A trap can also be triggered by instruction **ticc**. When these situations happens, the system sets the trap flag. Some control transfer instructions such as **bicca**, can cause the next instruction to be annulled under certain conditions. When this situation arises, the system sets the annulling flag. The way to process these 2 flags will be introduced in Sec 3.4.

Window registers We use the frame and the frame list to describe the window registers and window rotating. A frame f is an array that contains 8 words, and a frame list F is a list of frames. The definitions of them are given as follows:

$$(FrameList) \quad F \quad ::= \quad \mathbf{nil} \mid f :: F \qquad (Frame) \quad f \quad ::= \quad [w_0, \ldots, w_7]$$

Recall that we divide the general registers into four groups, global, out, local and in, as shown in Fig. 4(1), they represent the current view of the accessible general registers. There are also unaccessible registers, which are grouped into frames

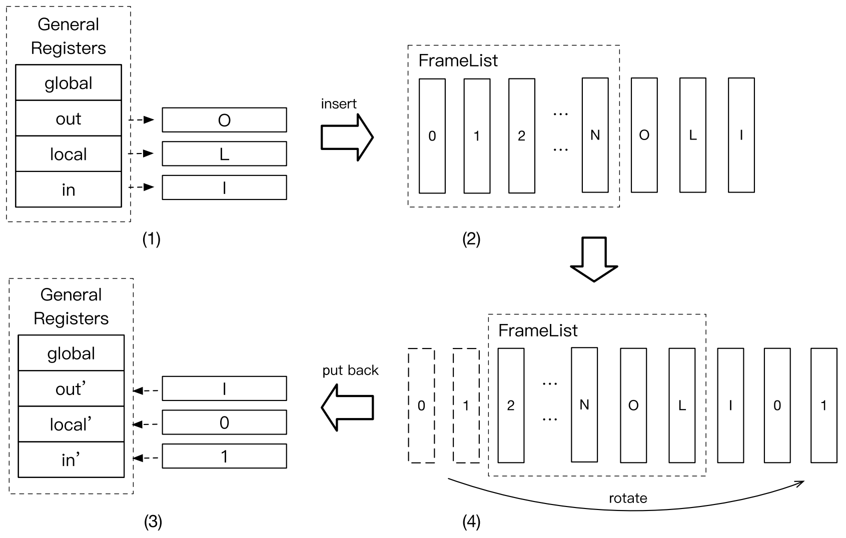

Fig. 4. Left rotation of the window.

$$
\begin{array}{lcl}
\text{left\_win}(Q) & \overset{def}{=\!=} & \textbf{let } (F', l) := \text{left}(F, \text{fetch}(R)) \textbf{ in} \\
 & & \textbf{let } R' := \text{replace}(l, R) \textbf{ in } (R'\{cwp \rightsquigarrow \text{post\_cwp}(R)\}, F') \\
 & & \quad \textbf{where } Q = (R, F) \\
\text{left}(F_L, F_I) & \overset{def}{=\!=} & (\ F'_L \mathbin{++} (p :: q :: \textbf{nil})\ ,\ F'_I \mathbin{++} (m :: n :: \textbf{nil})\ ) \\
 & & \quad \textbf{where } F_L = m :: n :: F'_L,\ F_I = p :: q :: F'_I \\
\text{right\_win}(Q) & \overset{def}{=\!=} & \textbf{let } (F', l) := \text{right}(F, \text{fetch}(R)) \textbf{ in} \\
 & & \textbf{let } R' := \text{replace}(l, R) \textbf{ in } (R'\{cwp \rightsquigarrow \text{pre\_cwp}(R)\}, F') \\
 & & \quad \textbf{where } Q = (R, F) \\
\text{right}(F_L, F_I) & \overset{def}{=\!=} & (\ (p :: q :: \textbf{nil}) \mathbin{++} F'_L\ ,\ (m :: n :: \textbf{nil}) \mathbin{++} F'_I\ ) \\
 & & \quad \textbf{where } F_L = F'_L \mathbin{++} (m :: n :: \textbf{nil})\ ,\ F_I = F'_I \mathbin{++} (p :: q :: \textbf{nil}) \\
\text{fetch}(R) & \overset{def}{=\!=} & R[r_8, \ldots, r_{15}] :: R[r_{16}, \ldots, r_{23}] :: R[r_{24}, \ldots, r_{31}] :: \textbf{nil} \\
\text{replace}(l, R) & \overset{def}{=\!=} & R\{[r_8, \ldots, r_{15}] \rightsquigarrow f_o\}\{[r_{16}, \ldots, r_{24}] \rightsquigarrow f_l\}\{[r_{24}, \ldots, r_{31}] \rightsquigarrow f_i\} \\
 & & \quad \textbf{where } l = f_o :: f_l :: f_i :: \textbf{nil} \\
\text{post\_cwp}(R) & \overset{def}{=\!=} & (R(cwp) + 1) \textbf{ mod } N \\
\text{pre\_cwp}(R) & \overset{def}{=\!=} & (R(cwp) - 1 + N) \textbf{ mod } N \\
R[r_i, \ldots, r_{i+7}] & \overset{def}{=\!=} & [R(r_i), \ldots, R(r_{i+7})] \\
R\{[r_i, \ldots, r_{i+7}] \rightsquigarrow f\} & \overset{def}{=\!=} & R\{r_i \rightsquigarrow w_0\} \ldots \{r_{i+7} \rightsquigarrow w_7\} \quad \textbf{where } f = [w_0, \ldots, w_7]
\end{array}
$$

Fig. 5. Definitions of the window rotation operation.

and stored in the frame list. We pair the register file and the frame list together as the register state Q, then we can define some window rotation operations on it, as shown below.

$$
\text{inc\_win}(Q) \overset{def}{=\!=} \begin{cases} \text{left\_win}(Q) & \textbf{if } \neg\text{win\_invalid}(\text{post\_cwp}(R), R) \\ \bot & \textbf{otherwise} \end{cases}
$$

$$
\text{dec\_win}(Q) \overset{def}{=\!=} \begin{cases} \text{right\_win}(Q) & \textbf{if } \neg\text{win\_invalid}(\text{pre\_cwp}(R), R) \\ \bot & \textbf{otherwise} \end{cases}
$$

$$
\text{set\_win}(w, Q) \overset{def}{=\!=} \begin{cases} \text{set\_win}(w, \text{left\_win}(Q)) & \textbf{if } w \neq R(cwp) \\ Q & \textbf{otherwise} \end{cases}
$$
$$
\textbf{where } Q = (R, F),\ \text{win\_invalid}(w, R) \overset{def}{=\!=} 2^w \ \&\&\ R(wim) \neq 0
$$

inc_win (or dec_win) operation rotates the window to the left (or right) if the next window is valid (¬win_invalid). set_win operation rotates the window continuously, until the current window's label is w. These three operations are based on the left rotation and right rotation operations. These two operations are symmetrical, so we focus on one of them - below we

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

$$\mathsf{exe\_delay}(Q, D) \stackrel{def}{=\!=} \begin{cases} \textbf{let } R' ::= \mathsf{dwrite\_psr}(w, R) \textbf{ in} & \\ (\mathsf{set\_win}(w_{<4:0>}, (R', F)), D') & \textbf{if } D = (0, psr, w) :: D' \\ \textbf{let } R' ::= \mathsf{dwrite\_tbr}(w, R) \textbf{ in} & \\ ((R', F)), D') & \textbf{if } D = (0, tbr, w) :: D' \\ \textbf{let } R' ::= \mathsf{dwrite\_wim}(w, R) \textbf{ in} & \\ ((R', F)), D') & \textbf{if } D = (0, wim, w) :: D' \\ ((R\{\varsigma \rightsquigarrow w\}, F), D') & \textbf{if } D = (0, \varsigma, w) :: D', \\ & \quad \varsigma \neq psr \textbf{ or } tbr \textbf{ or } wim \\ \textbf{let } (Q', D'') ::= \mathsf{exe\_delay}(Q, D') \textbf{ in} & \\ (Q', (n-1, \varsigma, w) :: D'') & \textbf{if } D = (n, \varsigma, w) :: D', n > 0 \\ (Q, D) & \textbf{otherwise} \end{cases}$$

where $Q = (R, F)$,

$$\mathsf{dwrite\_psr}(w, R) \stackrel{def}{=\!=} R\{n \rightsquigarrow w_{<23>}\}\{z \rightsquigarrow w_{<22>}\}\{v \rightsquigarrow w_{<21>}\}\{c \rightsquigarrow w_{<20>}\}\{s \rightsquigarrow w_{<7>}\}\{ps \rightsquigarrow w_{<6>}\},$$

$$\mathsf{dwrite\_tbr}(w, R) \stackrel{def}{=\!=} R\{tba \rightsquigarrow w_{<31:12>}\},$$

$$\mathsf{dwrite\_wim}(w, R) \stackrel{def}{=\!=} R\{wim_{<(N-1):0>} \rightsquigarrow w_{<(N-1):0>}\}$$

Fig. 6. Definitions of delayed writes.

demonstrate the left rotation operation in Fig. 4, the formal definition of it is given as left_win(Q) in Fig. 5. The rotation takes the following steps:

- We convert three groups of the general registers (out, local and in) into a frame list consisting of 3 frames as shown in Fig. 4(1). The conversion is formalized as fetch(R) in Fig. 5.
- As shown in Fig. 4(2) and (4), we can insert these 3 frames at the end of the frame list, then rotate the frame list to the left.
- Finally, as shown in Fig. 4(4) and (3), we remove 3 frames from the tail of the frame list, and insert them to the corresponding positions in the 32 general registers. The last two steps are modeled as $(F', l) := \mathsf{left}(F, \mathsf{fetch}(R))$ and $R' := \mathsf{replace}(l, R)$ in Fig. 5.

3.3.2. Delayed writes

When the system executes the **wr** instruction to write the symbol register, the execution will be delayed for X cycles. The value of X is implementation-dependent ($0 \le X \le 3$). The delay list D consists of a sequence of delayed writes d. Each d is a triple consisting of the remaining cycles to be delayed, the target register and the value to be written.

$$\begin{array}{rlll} (\mathit{DelayList}) & D & ::= & \textbf{nil} \mid d :: D \\ (\mathit{DelayItem}) & d & ::= & (c, \varsigma, w) \end{array} \qquad \begin{array}{rlll} (\mathit{DelayCycle}) & c & \in & [0..X] \\ (\mathit{InitDC}) & X & \in & [0..3] \end{array}$$

There are 2 operations defined on the delay list, as shown in Fig. 6. When the system executes the **wr** instruction, it will insert a delayed write into the delay list using function set_delay. At the beginning of each instruction cycle, the system scans the delay list, removes the delayed writes whose delay cycles are 0 and executes them, and then decrements the delay cycles of the remaining delayed writes. Notice that the instruction **wr** *psr* immediately writes the ET and PIL fields for interrupts. The lowest 12 bits of the register *tbr* are always 0, and the bits higher than N of *wim* are not used, so we ignore these fields here.

3.4. Operational semantics

As we mention in Sec.3.1, we define the operational semantics with multiple layers as shown in Fig. 1, where the main features of SPARCv8 are introduced at different layers. Next, we introduce the key operational semantics rules of each layer, from bottom to top. The omitted rules can be found in the Coq implementations [9].

3.4.1. Interrupts, traps and mode switch

In each instruction cycle, the system deals with interrupts and traps first. These rules are shown in Fig. 7.

- The INTERRUPT rule: If there is an interrupt request with level w and it is accepted (interrupt), the system triggers a trap after this external interrupt happens. It notes the trap type (get_tt) as w' and executes this trap (exe_trap), then dispatches an instruction.
  The function interrupt is to determine whether a interrupt is allowed. If the interrupt request satisfies the condition (the system doesn't have any trap now (¬has_trap), it allows traps to occur (trap_enabled), and the level of the interrupt

$$\boxed{\Delta \vdash S \xRightarrow{e} S'}$$

$$\frac{\begin{array}{c}\text{interrupt}(w, Q) = Q' \quad \text{get\_tt}(Q') = w' \quad \text{exe\_trap}(Q') = Q'' \\ C_s \vdash (M_s, Q'', D) \bullet\!\!\longrightarrow (M_s', Q''', D')\end{array}}{(C_u, C_s) \vdash ((M_u, M_s), Q, D) \xRightarrow{w'} ((M_u, M_s'), Q''', D')} \quad (\text{INTERRUPT})$$

$$\frac{\begin{array}{c}\text{has\_trap}(Q) \quad \text{get\_tt}(Q) = w \quad \text{exe\_trap}(Q) = Q' \\ C_s \vdash (M_s, Q', D) \bullet\!\!\longrightarrow (M_s', Q'', D')\end{array}}{(C_u, C_s) \vdash ((M_u, M_s), Q, D) \xRightarrow{w} ((M_u, M_s'), Q'', D')} \quad (\text{EXE-TRAP})$$

$$\frac{\neg\text{has\_trap}(Q) \quad \text{usr\_mode}(Q) \quad C_u \vdash (M_u, Q, D) \bullet\!\!\longrightarrow (M_u', Q', D')}{(C_u, C_s) \vdash ((M_u, M_s), Q, D) \Longrightarrow ((M_u', M_s), Q', D')} \quad (\text{EXE-USR})$$

$$\frac{\neg\text{has\_trap}(Q) \quad \text{sup\_mode}(Q) \quad C_s \vdash (M_s, Q, D) \bullet\!\!\longrightarrow (M_s', Q', D')}{(C_u, C_s) \vdash ((M_u, M_s), Q, D) \Longrightarrow ((M_u, M_s'), Q', D')} \quad (\text{EXE-SUP})$$

$$\frac{\text{interrupt}(w, Q) = Q' \quad \text{exe\_trap}(Q') = \bot}{(C_u, C_s) \vdash ((M_u, M_s), Q, D) \Longrightarrow \textbf{abort}} \qquad \frac{\text{has\_trap}(Q) \quad \text{exe\_trap}(Q) = \bot}{(C_u, C_s) \vdash ((M_u, M_s), Q, D) \Longrightarrow \textbf{abort}}$$

$$\frac{\text{interrupt}(w, Q) = Q' \quad \text{exe\_trap}(Q') = Q'' \quad C_s \vdash (M_s, Q'', D) \bullet\!\!\longrightarrow \textbf{abort}}{(C_u, C_s) \vdash ((M_u, M_s), Q, D) \Longrightarrow \textbf{abort}}$$

$$\frac{\text{has\_trap}(Q) \quad \text{exe\_trap}(Q) = Q' \quad C_s \vdash (M, Q', D) \bullet\!\!\longrightarrow \textbf{abort}}{(C_u, C_s) \vdash ((M_u, M_s), Q, D) \Longrightarrow \textbf{abort}}$$

$$\frac{\neg\text{has\_trap}(Q) \quad \text{usr\_mode}(Q) \quad C_u \vdash (M_u, Q, D) \bullet\!\!\longrightarrow \textbf{abort}}{(C_u, C_s) \vdash ((M_u, M_s), Q, D) \Longrightarrow \textbf{abort}}$$

$$\frac{\neg\text{has\_trap}(Q) \quad \text{sup\_mode}(Q) \quad C_s \vdash (M_s, Q, D) \bullet\!\!\longrightarrow \textbf{abort}}{(C_u, C_s) \vdash ((M_u, M_s), Q, D) \Longrightarrow \textbf{abort}}$$

Fig. 7. The rule of interrupts, traps and mode switch.

$$\begin{array}{lll}
\text{interrupt}(w, Q) & \stackrel{def}{=\!=} & \begin{cases} \text{set\_trap}(R\{tt \rightsquigarrow 16 + w\}) & \textbf{if } \neg\text{has\_trap}(R),\ \text{trap\_enabled}(R), \\ & \ 1 \le w \le 15,\ \ w = 15 \vee R(pil) < w \\ \bot & \textbf{otherwise} \end{cases} \\
& & \textbf{where } Q = (R, F) \\
\text{get\_tt}(Q) & \stackrel{def}{=\!=} & R(tt) \ \ \textbf{where } Q = (R, F) \\
\text{exe\_trap}(Q) & \stackrel{def}{=\!=} & \begin{cases} \textbf{let } (R', F') ::= \text{right\_win}(Q) \textbf{ in} & \textbf{if } \text{trap\_enabled}(R) \\ \textbf{let } R'' ::= \text{to\_sup}(\text{save\_mode}(\text{disable\_trap}(R'))) \textbf{ in} & \\ (\text{tbr\_jmp}(\text{clear\_trap}(\text{save\_pc\_npc}(r_{17}, r_{18}, R''))), F') & \\ \bot & \textbf{otherwise} \end{cases} \\
& & \textbf{where } Q = (R, F) \\
\text{save\_pc\_npc}(r_m, r_n, R) & \stackrel{def}{=\!=} & \begin{cases} R\{r_m \rightsquigarrow R(pc)\}\{r_n \rightsquigarrow R(npc)\} & \textbf{if } \neg\text{annulled}(R) \\ \text{clear\_annul}(R\{r_m \rightsquigarrow R(npc)\} & \textbf{otherwise} \\ \qquad\qquad \{r_n \rightsquigarrow R(npc + 4)\}) & \end{cases}
\end{array}$$

$$\begin{array}{lll@{\qquad\qquad}lll}
\text{set\_trap}(R) & \stackrel{def}{=\!=} & R\{\tau \rightsquigarrow 1\} & \text{has\_trap}(R) & \stackrel{def}{=\!=} & R(\tau) \neq 0 \\
\text{clear\_trap}(R) & \stackrel{def}{=\!=} & R\{\tau \rightsquigarrow 0\} & \text{trap\_enabled}(R) & \stackrel{def}{=\!=} & R(et) \neq 0 \\
\text{disable\_trap}(R) & \stackrel{def}{=\!=} & R\{et \rightsquigarrow 0\} & \text{annulled}(R) & \stackrel{def}{=\!=} & R(\kappa) \neq 0 \\
\text{clear\_annul}(R) & \stackrel{def}{=\!=} & R\{\kappa \rightsquigarrow 0\} & \text{usr\_mode}(R) & \stackrel{def}{=\!=} & R(s) = 0 \\
\text{to\_sup}(R) & \stackrel{def}{=\!=} & R\{s \rightsquigarrow 1\} & \text{sup\_mode}(R) & \stackrel{def}{=\!=} & R(s) \neq 0 \\
\text{save\_mode}(R) & \stackrel{def}{=\!=} & R\{ps \rightsquigarrow R(s)\} & & &
\end{array}$$

Fig. 8. Auxiliary definitions for the rule of interrupts, traps and mode switch.

request is greater than the threshold or it is the highest level), it will set the trap flag (set_trap) and record the trap type in the *tt* field. The trap type can be read by function get_tt, as shown in Fig. 8.

When the system executes a trap (exe_trap), first it needs to make sure that the system allows traps to occur (trap_enabled). Then it rotates the window to the right (right_win) to save the current context, forbid traps to occur (disable_trap), save the current mode (save_mode) and switch to the supervisor mode (to_sup). Finally it saves the values of *pc* and *npc* to the register r_{17} and r_{18} (save_pc_npc), unset the trap flag (clear_trap) and jump to the address of the trap handler (tbr_jmp), as shown in Fig. 8. Function right_win and tbr_jmp are defined in Sec 3.3.

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

$$\frac{\text{exe\_delay}(Q, D) = (Q', D') \qquad \neg\text{annulled}(Q') \qquad C(Q'(pc)) = i \qquad (M, Q', D') \circ\!\!\xrightarrow{i} (M', Q'', D'')}{C \vdash (M, Q, D) \bullet\!\!\longrightarrow (M', Q'', D'')} \quad (\text{EXE-INS})$$

$$\frac{\text{exe\_delay}(Q, D) = (Q', D') \qquad \text{annulled}(Q') \qquad \text{next}(\text{clear\_annul}(Q')) = Q''}{C \vdash (M, Q, D) \bullet\!\!\longrightarrow (M, Q'', D')} \quad (\text{ANNULLED})$$

$$\frac{\text{exe\_delay}(Q, D) = (Q', D') \qquad \neg\text{annuled}(Q') \qquad C(Q'(pc)) = \bot}{C \vdash (M, Q, D) \bullet\!\!\longrightarrow \textbf{abort}}$$

$$\frac{\text{exe\_delay}(Q, D) = (Q', D') \qquad \neg\text{annuled}(Q') \qquad C(Q'(pc)) = i \qquad (M, Q, D) \circ\!\!\xrightarrow{i} \textbf{abort}}{C \vdash (M, Q, D) \bullet\!\!\longrightarrow \textbf{abort}}$$

Fig. 9. The rule of executing delay and handling annulling flag.

- The EXE-TRAP rule: If the system has a trap (has_trap), it will note the trap type (get_tt) and execute this trap (exe_trap), then dispatch an instruction.
- The EXE-USR rule and EXE-SUP rule: If the system does not have traps (¬has_trap), it will select the code heap and the memory according to the mode (usr_mode or sup_mode) then dispatch an instruction.

Besides these 3 rules, the system may **abort** if there is an exception when executing the trap or dispatching instructions, as shown in Fig. 7.

3.4.2. *Executing delay and handling annulling flag*

After dealing with interrupts and traps, the system will execute delay and handle the annulling flag. These rules are given in Fig. 9.

- The EXE-INS rule: It first executes the delayed writes (exe_delay, described in Sec. 3.3.2), then if the annulled flag has not been set (¬annulled), it will pick up an instruction at pc from the code heap and execute it. Function annulled is given in Fig. 8.
- The ANNULLED rule: After executing the delayed writes (exe_delay), if the annulled flag has been set (annulled), it will skip one instruction and unset the annulling flag (clear_annul). Function clear_annul and annulled are given in Fig. 8. Function next is defined in Sec 3.3.1.

Besides these 2 rules, the system may **abort** if it can not fetch an instruction at pc from code heap or there is an exception when dispatching instructions, these roles can be found in Fig. 9.

3.4.3. *Instruction dispatch*

When the system dispatches a instruction, it may only access the register file R and the memory M. This kind of instruction is classified as *simple instructions*, as shown in Fig. 10 and Fig. 11.

- The **bicc** η β instruction evaluates the address expression β to get the value w, and requires the address w to be *word-aligned*. It decides whether to transfer by the conditional expression η. If the value of the conditional expression is *true*, it executes the delayed transfer (rule BICC-TRUE). Otherwise it makes no transfer (rule BICC-FALSE). Function djmp and next are given in Sec 3.3.1. Function word_aligned is given in Fig. 12.
- Besides the requirement of being *word-aligned* and whether to transfer according to the value of the conditional expression η, the **bicca** η β instruction also decides whether to be annulled by the conditional expression. If the value of the conditional expression is *false*, it makes no transfer and *sets the annulling flag* (set_annul) only (rule BICC-FALSE); if the type of the conditional expression is not *al* and the value is *true*, it executes the delayed transfer but does not annul the next instruction (rule BICC-TRUE); if the type of the conditional expression is *al* (the value of expression *al* is always $true$), it executes the delayed transfer and *sets the annulling flag* (rule BICC-AL). Function set_annul is given in Fig. 12.
- The **jmpl** β r_d instruction requires the value of expression β is *word-aligned*. It saves the current value of register pc to the register r_d (save_pc), and then executes the delayed transfer (rule JMPL). Function save_pc is given in Fig. 12.
- The **nop** instruction does nothing (rule NOP).
- The **ld** β r_d (or **st** r_d β) instruction requires the value of expression β is *word-aligned* and in the domain of M. Then it loads (or stores) the value of $M(w)$ into the register r_d (rules LD and ST).

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

$$\frac{[\![\, \beta \,]\!]_R = w \qquad \text{word\_aligned}(w) \qquad [\![\, \eta \,]\!]_R = true}{(M, R) \xrightarrow{\textbf{bicc}\ \eta\ \beta} (M, \text{djmp}(w, R))} \quad (\text{BICC-TRUE})$$

$$\frac{[\![\, \beta \,]\!]_R = w \qquad \text{word\_aligned}(w) \qquad [\![\, \eta \,]\!]_R = false}{(M, R) \xrightarrow{\textbf{bicc}\ \eta\ \beta} (M, \text{next}(R))} \quad (\text{BICC-FALSE})$$

$$\frac{[\![\, \beta \,]\!]_R = w \qquad \text{word\_aligned}(w) \qquad [\![\, \eta \,]\!]_R = false}{(M, R) \xrightarrow{\textbf{bicca}\ \eta\ \beta} (M, \text{set\_annul}(\text{next}(R)))} \quad (\text{BICCA-FALSE})$$

$$\frac{[\![\, \beta \,]\!]_R = w \qquad \text{word\_aligned}(w) \qquad \eta \neq al \qquad [\![\, \eta \,]\!]_R = true}{(M, R) \xrightarrow{\textbf{bicca}\ \eta\ \beta} (M, \text{djmp}(w, R))} \quad (\text{BICCA-TRUE})$$

$$\frac{[\![\, \beta \,]\!]_R = w \qquad \text{word\_aligned}(w)}{(M, R) \xrightarrow{\textbf{bicca}\ al\ \beta} (M, \text{set\_annul}(\text{djmp}(w, R)))} \quad (\text{BICCA-AL})$$

$$\frac{[\![\, \beta \,]\!]_R = w \qquad \text{word\_aligned}(w) \qquad \text{save\_pc}(r_d, R) = R'}{(M, R) \xrightarrow{\textbf{jmpl}\ \beta\ r_d} (M, \text{djmp}(w, R'))} \quad (\text{JMPL})$$

$$\frac{}{(M, R) \xrightarrow{\textbf{nop}} (M, \text{next}(R))} \quad (\text{NOP})$$

Fig. 10. Simple instructions (1).

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

$$\frac{[\![\, \beta \,]\!]_R = w \qquad \text{word\_aligned}(w) \qquad w \in dom(M) \qquad R' = R\{r_d \leadsto M(w)\}}{(M, R) \xrightarrow{\textbf{ld}\ \beta\ r_d} (M, \text{next}(R'))} \quad (\text{LD})$$

$$\frac{[\![\, \beta \,]\!]_R = w \qquad \text{word\_aligned}(w) \qquad w \in dom(M) \qquad M' = M\{w \leadsto [\![\, r_d \,]\!]_R\}}{(M, R) \xrightarrow{\textbf{st}\ r_d\ \beta} (M', \text{next}(R))} \quad (\text{ST})$$

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

$$\frac{[\![\, \gamma \,]\!]_R = w \qquad [\![\, \eta \,]\!]_R = true}{(M, R) \xrightarrow{\textbf{ticc}\ \eta\ \gamma} (M, \text{set\_user\_trap}(w_{<6:0>}, R))} \quad (\text{TICC-TRUE})$$

$$\frac{\text{sup\_mode}(R) \qquad R' = R\{r_d \leadsto R(\varsigma)\}}{(M, R) \xrightarrow{\textbf{rd}\ \varsigma\ r_d} (M, \text{next}(R'))} \quad (\text{RD-SUP})$$

$$\frac{\text{usr\_mode}(R) \qquad \varsigma = y\ \textbf{or}\ asr_i \qquad R' = R\{r_d \leadsto R(\varsigma)\}}{(M, R) \xrightarrow{\textbf{rd}\ \varsigma\ r_d} (M, \text{next}(R'))} \quad (\text{RD-USR})$$

$$\frac{[\![\, \alpha \,]\!]_R = w \qquad r_s \ \&\&\ w = w' \qquad R' = R\{r_d \leadsto w'\}}{(M, R) \xrightarrow{\textbf{and}\ r_s\ \alpha\ r_d} (M, \text{next}(R'))} \quad (\text{AND})$$

Fig. 11. Simple instructions (2).

$$\begin{aligned}
&\mathsf{save\_pc}(r_i, R) \overset{def}{=\!=} R\{r_i \rightsquigarrow R(pc)\}\\
&\mathsf{set\_user\_trap}(k, R) \overset{def}{=\!=} \mathsf{set\_trap}(R\{tt \rightsquigarrow 128 + k\})\\
&\mathsf{rett\_f}(Q) \overset{def}{=\!=} \begin{cases} (\mathsf{restore\_mode}(\mathsf{enable\_trap}(R')), F') & \textbf{if } \mathsf{inc\_win}(Q) = (R', F') \\ \bot & \textbf{otherwise} \end{cases}\\
&\mathsf{word\_aligned}(w) \overset{def}{=\!=} w_{<1:0>} = 0 \qquad \mathsf{enable\_trap}(R) \overset{def}{=\!=} R\{et \rightsquigarrow 1\}\\
&\mathsf{restore\_mode}(R) \overset{def}{=\!=} R\{s \rightsquigarrow R(ps)\} \qquad \mathsf{set\_annul}(R) \overset{def}{=\!=} R\{\kappa \rightsquigarrow 1\}
\end{aligned}$$

Fig. 12. Auxiliary definitions for the rule of instructions.

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

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

$$\frac{\mathsf{dec\_win}(R, F) = (R', F') \qquad [\![\alpha]\!]_R = a \qquad R'' = R'\{r_d \rightsquigarrow [\![r_s]\!]_R + a\}}{(M, (R, F), D) \circ\!\!\xrightarrow{\textbf{save}\ r_s\ \alpha\ r_d} (M, (\mathsf{next}(R''), F'), D)} \quad (\text{SAVE})$$

$$\frac{\mathsf{inc\_win}(R, F) = (R', F') \qquad [\![\alpha]\!]_R = a \qquad R'' = R'\{r_d \rightsquigarrow [\![r_s]\!]_R + a\}}{(M, (R, F), D) \circ\!\!\xrightarrow{\textbf{restore}\ r_s\ \alpha\ r_d} (M, (\mathsf{next}(R''), F'), D)} \quad (\text{RESTORE})$$

$$\frac{\begin{array}{c}\neg\mathsf{trap\_enabled}(R) \qquad \mathsf{sup\_mode}(R) \qquad [\![\beta]\!]_R = w \\ \mathsf{word\_aligned}(w) \qquad \mathsf{rett\_f}(R, F) = (R', F')\end{array}}{(M, (R, F), D) \circ\!\!\xrightarrow{\textbf{rett}\ \beta} (M, (\mathsf{djmp}(w, R'), F'), D)} \quad (\text{RETT})$$

Fig. 13. Complex instructions (1).

- The **ticc** $\eta\ \gamma$ instruction evaluates the trap expression γ. If the condition η is true, it *sets the trap flag* and records the trap type by using function set_user_rtap (rule TICC-TRUE), otherwise it does nothing (rule TICC-FALSE). The function set_user_rtap is given in Fig. 12, where the input parameter $w_{<6:0>}$ represents the lowest 7 bits of w.
- The **rd** $\varsigma\ r_d$ instruction loads the value of register ς into register r_d (rules RD-SUP and RD-USR). If the system is in user mode, the register ς must be y or asr_i.
- There are also some arithmetical instructions. Here we only give the **and** instruction as an example. The **and** $r_s\ \alpha\ r_d$ instruction evaluates the operand expression α to get the value w, then executes the *and* operation and writes the result into register r_d (rule AND).

Besides these *simple instructions*, there are also some *complex instructions* that involve windows registers and delayed writes, as shown in Fig. 13 and Fig. 14.

- We use the rule LIFT to lift the transition $(M, R) \xrightarrow{i} (M', R')$ to $(M,Q,D) \circ\!\!\xrightarrow{i} (M', Q', D')$.
- The instruction **save** $r_s\ \alpha\ r_d$ (or **restore** $r_s\ \alpha\ r_d$) first decreases (or increases) the label of the window, then it evaluates the operand expression α to get the value a and writes the result of expression $[\![r_s]\!]_R + a$ to the register r_d (rules SAVE and RESTORE). The definitions of function dec_win and inc_win can be found in Sec. 3.3.1.
- The instruction **rett** β requires that the system is in the supervisor mode and does not allow traps to occur, and the value of the address expression β is *word-aligned*. It increases the label of the window to restore the context (inc_win), allows traps to occur (enable_trap) and restores the previous mode (restore_mode) by using function rett_f which is defined in Fig. 12.
- When the **wr** $r_d\ \alpha\ \varsigma$ instruction is executed in the user mode, since it does not have the permission to access the register *wim*, *tbr* and *psr*, so ς must be y or asr_i (rule WR-USR); when it is executed in the supervisor mode, it has the permission to access all symbol registers (rule WR-SUP). Then it executes the *xor* operation, remembers the results as w, and inserts the triple (X, ς, w) into the delay list D. When the ς is *psr*, the ET and PIL fields are written immediately, with respect to interrupts (rule WR-PSR). The function set_delay is given in Sec 3.3.2.

Exceptions In the above rules, if some of the conditions (e.g., being *word-aligned*) are not satisfied, the system will throw exceptions. Exceptions include traps and system failure (i.e., the system aborts). When an instruction causes traps due to *memory not aligned*, *window overflow*, etc., it will record the trap type (rule GEN-TRAP in Fig. 14). Then the system will execute this trap in the next cycle, as explained in Sec 3.4.1. As shown in Fig. 15, the function unexpected_trap checks whether an instruction has a trap by using function trap_type. If that happens, it writes the trap type to the register *tt* and sets the trap

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

$$\frac{\begin{array}{ccc}\mathsf{usr\_mode}(R) & \varsigma = y \ \textbf{or}\ asr_i & [\![\, \alpha \,]\!]_R = a \\ [\![\, r_s \,]\!]_R \ \textbf{xor}\ a = w & & D' = \mathsf{set\_delay}(\varsigma, w, D)\end{array}}{(M, (R, F), D) \circ\!\!\xrightarrow{\textbf{wr}\ r_d\ \alpha\ \varsigma} (M, (\mathsf{next}(R), F), D')}\ (\textsc{wr-usr})$$

$$\frac{\begin{array}{ccc}\mathsf{sup\_mode}(R) & \varsigma \neq psr & [\![\, \alpha \,]\!]_R = a \\ [\![\, r_s \,]\!]_R \ \textbf{xor}\ a = w & & D' = \mathsf{set\_delay}(\varsigma, w, D)\end{array}}{(M, (R, F), D) \circ\!\!\xrightarrow{\textbf{wr}\ r_d\ \alpha\ \varsigma} (M, (\mathsf{next}(R), F), D')}\ (\textsc{wr-sup})$$

$$\frac{\begin{array}{cccc}\mathsf{sup\_mode}(R) & [\![\, \alpha \,]\!]_R = a & [\![\, r_s \,]\!]_R \ \mathtt{xor}\ a = w & w_{<4:0>} < N \\ D' = \mathsf{set\_delay}(psr, w, D) & & R' = R\{et \rightsquigarrow w_{<5>}\}\{pil \rightsquigarrow w_{<11:8>}\} &\end{array}}{(M, (R, F), D) \circ\!\!\xrightarrow{\textbf{wr}\ r_s\ \alpha\ psr} (M, (\mathsf{next}(R'), F), D')}\ (\textsc{wr-psr})$$

$$\frac{\mathsf{unexpected\_trap}(i, Q) = Q'}{(M, Q, D) \circ\!\!\xrightarrow{i} (M, Q', D)}\ (\textsc{gen-trap}) \qquad \frac{\mathsf{abort\_ins}(i, Q, M)}{(M, Q, D) \circ\!\!\xrightarrow{i} \textbf{abort}}\ (\textsc{abort})$$

Fig. 14. Complex instructions (2).

$$\mathsf{unexpected\_trap}(i, Q) \stackrel{def}{=\!=} \begin{array}{l}\textbf{let}\ w = \mathsf{trap\_type}(i, Q)\ \textbf{in} \\ \begin{cases} (\mathsf{set\_trap}(R\{tt \rightsquigarrow w\}), F) & \textbf{if}\ w \neq \bot \\ \bot & \textbf{otherwise}\end{cases} \\ \textbf{where}\ Q = (R, F)\end{array}$$

$$\mathsf{trap\_type}(i, Q) \stackrel{def}{=\!=} \begin{array}{l}\begin{cases} privileged\_ins(3) & \textbf{if}\ i = \textbf{rd}\ \varsigma\ r_d\ \ \textbf{or}\ \ \textbf{wr}\ r_d\ \alpha\ \varsigma,\ [\![\, \alpha \,]\!]_R \neq \bot, \\ & \quad \mathsf{usr\_mode}(R), \varsigma = wim\ \textbf{or}\ tbr\ \textbf{or}\ psr \\ & \textbf{if}\ i = \textbf{rett}\ \beta,\ \mathsf{trap\_enabled}(R), \mathsf{usr\_mode}(R) \\ illegal\_ins(2) & \textbf{if}\ \ i = \textbf{wr}\ r_s\ \alpha\ psr,\ [\![\, \alpha \,]\!]_R = w, \\ & \quad ([\![\, r_s \,]\!]_R\ \textbf{xor}\ w)_{<4:0>} \geq N \\ & \textbf{if}\ i = \textbf{rett}\ \beta,\ \mathsf{trap\_enabled}(R), \mathsf{sup\_mode}(R) \\ win\_overflow(5) & \textbf{if}\ i = \textbf{save}\ r_s\ \alpha\ r_d,\ [\![\, \alpha \,]\!]_R \neq \bot, \mathsf{dec\_win}(Q) = \bot \\ win\_underflow(6) & \textbf{if}\ i = \textbf{restore}\ r_s\ \alpha\ r_d,\ [\![\, \alpha \,]\!]_R \neq \bot, \mathsf{inc\_win}(Q) = \bot \\ mem\_not\_align(7) & \textbf{if}\ i = \textbf{ld}\ \beta\ r_d\ \ \textbf{or}\ \ \textbf{st}\ r_d\ \beta\ \ \textbf{or jmpl}\ \beta\ r_d\ \ \textbf{or} \\ & \quad \textbf{bicc}\ \eta\ \beta,\ [\![\, \beta \,]\!]_R = w, \neg\mathsf{word\_aligned}(w) \\ \ldots & \ldots \\ \bot & \textbf{otherwise}\end{cases} \\ \textbf{where}\ Q = (R, F)\end{array}$$

Fig. 15. The definition of traps.

flag. The function trap_type judges whether there is a trap caused by instructions. If the instruction causes i a trap, it will return the trap type, The details are as follows:

- It will cause a *privileged_instruction* trap if the **rd** (or **wr**) instruction attempts to access the *wim*, *psr*, and *tbr* registers in the user mode, or the **rett** instruction is executed when the system allows traps to occur in the user mode.
- An *illegal_ins* trap is generated if the **rett** instruction is executed when the system allows traps to occur in the supervisor mode, or the new value of *cwp* given by the instruction **wr** is out of range.
- The *win_overflow* (or *win_underflow*) trap is occurred if the next window is invalid when executing the **save** (or **restore**) instruction.
- A *mem_not_align* trap is occurred if the address in the instruction **ld**, **st**, **jmpl** or **bicc** is not *word-aligned*.

In the ABORT rule in Fig. 14, the function unexpected_trap checks whether an instruction can cause the system to abort. As shown in Fig. 16, the system aborts if one of these conditions holds:

- If the immediate value of expression α, β or γ is out of range.
- The address in the **ld** (or **st**) instruction is not in the domain of the memory.
- When the system executes the **rett** instruction to return from a trap, it will abort if it is in the user mode, or the address in the instruction is not *word-aligned*, or the next window is invalid when it tries to rotate the window to restore the context.

$$\mathsf{abort\_ins}(i, Q, M) \overset{def}{=\!=} \begin{cases} true & \textbf{if } i = \textbf{ld}\ \beta\ r_d \ \textbf{ or } \ \textbf{st}\ r_d\ \beta \ \textbf{ or jmpl}\ \beta\ r_d \ \textbf{ or} \\ & \quad \textbf{rett}\ \beta,\ [\![\beta]\!]_R = \bot \\ true & \textbf{if } i = \textbf{ld}\ \beta\ r_d \ \textbf{ or } \ \textbf{st}\ r_d\ \beta,\ [\![\beta]\!]_R = w, w \notin dom(M) \\ true & \textbf{if } i = \textbf{udivcc}\ r_s\ \alpha\ r_d \ \textbf{ or } \ \textbf{save}\ r_s\ \alpha\ r_d \ \textbf{ or} \\ & \quad\quad \textbf{restore}\ r_s\ \alpha\ r_d \ \textbf{ or } \ \textbf{wr}\ r_d\ \alpha\ \varsigma,\ [\![\alpha]\!]_R = \bot \\ true & \textbf{if } i = \textbf{ticc}\ \eta\ \gamma,\ [\![\gamma]\!]_R = \bot \\ true & \textbf{if } i = \textbf{rett}\ \beta,\ [\![\beta]\!]_R = w, \\ & \quad \neg\mathsf{word\_aligned}(w) \vee \mathsf{usr\_mode}(R) \vee \mathsf{inc\_win}(Q) = \bot \\ \ldots & \ldots \\ false & \textbf{otherwise} \end{cases}$$
$$\textbf{where } Q = (R, F)$$

Fig. 16. The definition of abort.

3.4.4. Multi-step execution

In a single step, the system changes from the state S to the state S' and produces an event e. The event e is used to record whether the system has a trap in an instruction cycle.

$$(Event)\ \ e\ ::=\ w \mid \bot \qquad (EventList)\ \ E\ ::=\ \textbf{nil} \mid e :: E$$

If a trap occurs, the corresponding trap type w is recorded as an event, otherwise it is $\bot$. The transition of zero-or-multiple steps is defined as below. Multiple steps generate multiple events, namely the event list E.

$$\frac{}{\Delta \vdash S \overset{\textbf{nil}}{\Longmapsto}{}^{0}\ S} \qquad \frac{\Delta \vdash S \overset{e}{\Longrightarrow} S'' \qquad \Delta \vdash S'' \overset{E}{\Longmapsto}{}^{n}\ S'}{\Delta \vdash S \overset{e::E}{\Longmapsto}{}^{n+1}\ S'}$$

4. Determinacy and isolation properties

In this section, we will prove that our formal model satisfies the determinacy and isolation properties. The determinacy property explains that the execution of the machine is deterministic with the given sequence of external interrupts. The isolation property characterizes separation of the memory space of the user mode and the supervisor mode, which guarantees the space security of the entire system.

We use $\Delta \vdash S \overset{E}{\Longmapsto}{}^{*}\ S'$ to represent zero-or-multiple steps of the execution under the given sequence of external interrupts E. Theorem 1 says that, if two executions start from the same initial states and both of them produce the same sequence of external interrupts, then they should arrive at the same final states.

Theorem 1. (Determinacy)
If $\Delta \vdash S \overset{E}{\Longmapsto}{}^{*}\ S_1$, $\Delta \vdash S \overset{E}{\Longmapsto}{}^{*}\ S_2$, *then* $S_1 = S_2$. *where* $\Delta \vdash S \overset{E}{\Longmapsto}{}^{*}\ S'$ *is defined as* $\exists n, \Delta \vdash S \overset{E}{\Longmapsto}{}^{n}\ S'$.

In SPARCv8 ISA, triggering a trap is the only way of switching to the supervisor mode. We will prove this property first. That is, if a system is running in the user mode at the beginning, it will run in the user mode forever if there is no trap. First, we give the conditions of running n steps in user mode as below:

$$\begin{array}{lll} \Delta \vdash S \bullet\!\!\Longrightarrow^{n} S' & \overset{def}{=\!=} & \mathsf{usr\_mode}(S)\ \wedge\ \mathsf{no\_delay\_item}(S)\ \wedge\ \Delta \vdash S \overset{E}{\Longmapsto}{}^{n}\ S' \\ & & \wedge\ \mathsf{no\_trap\_event}(E) \\ \mathsf{no\_delay\_item}(S) & \overset{def}{=\!=} & D = \textbf{nil} \qquad \textbf{where } S = ((M_u, M_s), Q, D) \\ \mathsf{no\_trap\_event}(E) & \overset{def}{=\!=} & \forall e \in E, e = \bot \end{array}$$

We first require the system to be in the user mode initially (usr_mode). Second, because of the delayed write feature, we need to require the delay list to be empty (no_delay_item), otherwise the system may enter the supervisor mode if there is a delayed write item in the delay list that will modify the S field of PSR. Finally, we require there is no trap in the system after several steps (no_trap_event).

After giving these conditions, we need to prove that the system is always running in the user mode under these conditions, as shown in Theorem 2:

Theorem 2. (In user mode) *If* $\Delta \vdash S \bullet\!\!\Longrightarrow^{n} S'$, *then* usr_mode$(S')$.

It says that, if the system satisfies all the conditions defined in $\Delta \vdash S \bullet\!\!\Longrightarrow^{n} S'$, it will be in the user mode after n steps. Since this theorem is true for all n, the system should be in the user mode after arbitrary steps. So we can call $\Delta \vdash S \bullet\!\!\Longrightarrow^{n} S'$ as "the system is running in the user mode for n-steps". This property will be used in proving the isolation property later.

Based on Theorem 2, we apply it to prove if a system is running in user mode, it does not have the permission to read and write the resource that belongs to the supervisor mode. The isolation property is shown below:

Theorem 3. (Write isolation)
If $\Delta \vdash S \bullet\!\!\Longrightarrow^n S'$, *then* $\mathsf{sup\_part\_eq}(S, S')$. $\mathsf{sup\_part\_eq}$ *is defined as:*

$$\begin{aligned}\mathsf{sup\_part\_eq}(S, S') \overset{def}{=\!=} \; & M_s = M_s' \\ & \textbf{where } S = ((M_u, M_s), Q, D)\ ,\ S' = ((M_u', M_s'), Q', D')\end{aligned}$$

Theorem 4. (Read isolation)
If $\mathsf{usr\_code\_eq}(\Delta_1, \Delta_2)$, $\mathsf{usr\_state\_eq}(S_1, S_2)$, *and* $\Delta_1 \vdash S_1 \bullet\!\!\Longrightarrow^n S_1'$, $\Delta_2 \vdash S_2 \bullet\!\!\Longrightarrow^n S_2'$, *then* $\mathsf{usr\_state\_eq}(S_1', S_2')$. $\mathsf{usr\_code\_eq}$ *and* $\mathsf{usr\_state\_eq}$ *are defined as:*

$$\begin{aligned}\mathsf{usr\_state\_eq}(S, S') \overset{def}{=\!=} \; & Q = Q' \wedge M_u = M_u' \\ & \textbf{where } S = ((M_u, M_s), Q, D)\ ,\ S' = ((M_u', M_s'), Q', D') \\ \mathsf{usr\_code\_eq}(\Delta, \Delta') \overset{def}{=\!=} \; & C_u = C_u' \\ & \textbf{where } \Delta = (C_u, C_s)\ ,\ \Delta' = (C_u', C_s')\end{aligned}$$

Theorem 3 shows that if the system is running in the user mode, it does not modify the resource that belongs to the supervisor mode. Theorem 4 shows that if a particular part of two systems are the same at the beginning, they will always be the same when the system is running in the user mode for several steps. The above two theorems show the isolation property of SPARCv8.

5. Verifying a window overflow trap handler

In this section, we will verify the correctness of the window overflow handler by showing that the handler can be safely executed under the given pre-condition. Here we only focus on the change of position of the invalid window, but omit the other properties, such as register-to-memory copies and the changes to other registers. The complete verification will be completed in the future work.

Because the window overflow and window underflow traps and their handlers are similar, here we only introduce window overflow and its handler. The details of verifying the window underflow trap handler can be found in the Coq implementations [9].

5.1. Window overflow

In SPARCv8, the wim register is used to distinguish whether the window is valid or not. A register window with label w is invalid if the w bit of register wim is 1. When we use the window registers, if we discard the local registers of one of these windows, the ring window will be cut off, and the remaining space can be treated as a stack. The bottom of the stack is the next of the discarded window, and from the current window to the bottom of the stack is the space that has been used, and the rest is the free space, as shown in Fig. 17.

When the **save** instruction needs to save the current context, it pushes the in and local registers of current window onto the stack, and the out registers become the next window's in registers. When the **restore** instruction needs to restore the context, it pops the in and local registers from the stack, as shown in Fig. 18.

The number of windows is finite. If the system executes a **save** instruction to save the context when all the windows have already been used, it will cause a window overflow trap. The window overflow trap handler will be executed to handle this trap. it will store the data that at the bottom of the stack into the memory, create a free space for the next **save** instruction. We give the details in the next subsection.

5.2. Dealing with window overflow

We give the code of the trap handler in Fig. 19. The window overflow trap is processed after the following steps:

- Fig. 20(a) shows that the system triggers a window overflow trap when it executes a **save** instruction while the next windows is invalid (wim is 1).
- After triggering a trap, the system will execute this trap (exe_trap), that is, rotate the window and jump to the trap handler, as shown in Fig. 20(b).
- The handler takes the next window as the invalid window, which is implemented by the cyclic shift operation (Lines 1-5 and 7-10), as shown in Fig. 20(c).

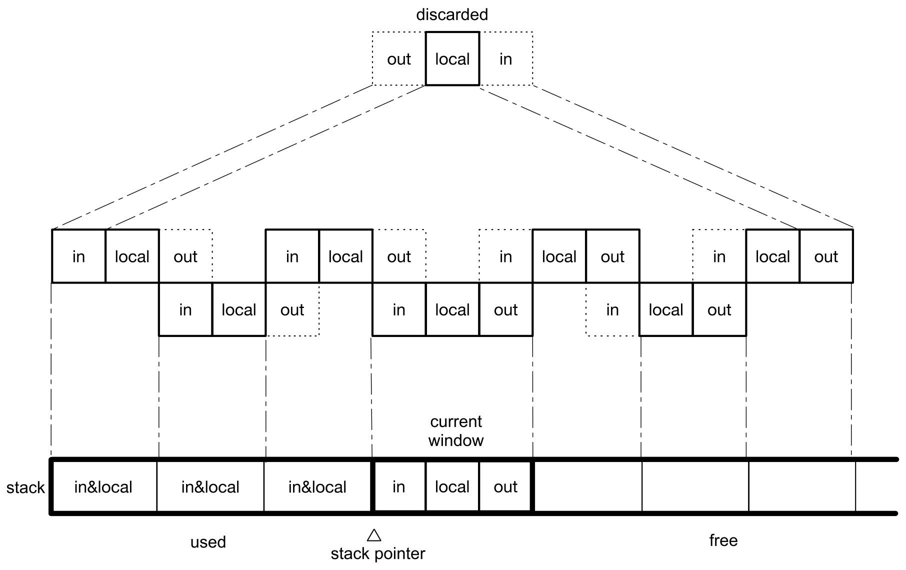

Fig. 17. Window registers are divided into the stack space and the current window.

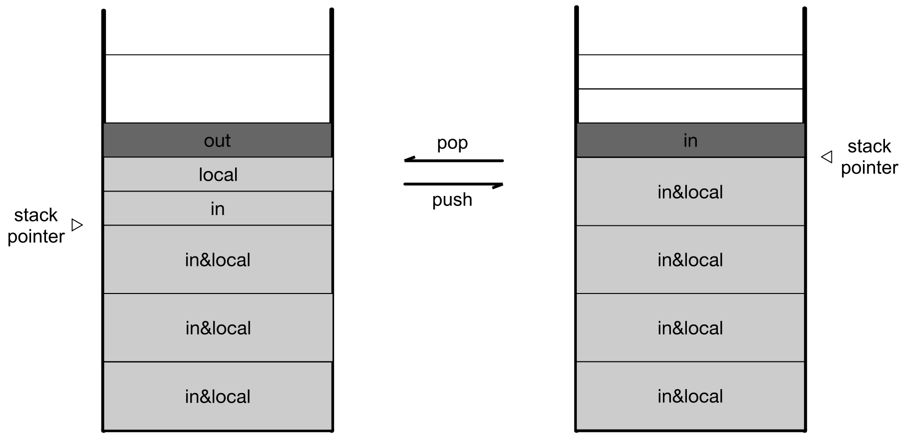

Fig. 18. The change of the stack when save and restore the context.

```
WINDOW OVERFLOW:
mov  %wim,%l3
mov  %g1,%l7
srl  %l3,1,%g1
sll  %l3,NWINDOWS-1,%l4
or %l4,%g1,%g1
save
mov  %g1,%wim
nop
nop
nop
st %l0,[%sp+0]
st %l1,[%sp+4]
st %l2,[%sp+8]
st %l3,[%sp+12]
st %l4,[%sp+16]
st %l5,[%sp+20]
st %l6,[%sp+24]
st %l7,[%sp+28]
st %i0,[%sp+32]
st %i1,[%sp+36]
st %i2,[%sp+40]
st %i3,[%sp+44]
st %i4,[%sp+48]
st %i5,[%sp+52]
st %i6,[%sp+56]
st %i7,[%sp+60]
restore
mov %l7,%g1
jmp %l1
rett %l2
```

Fig. 19. The window overflow trap handler.

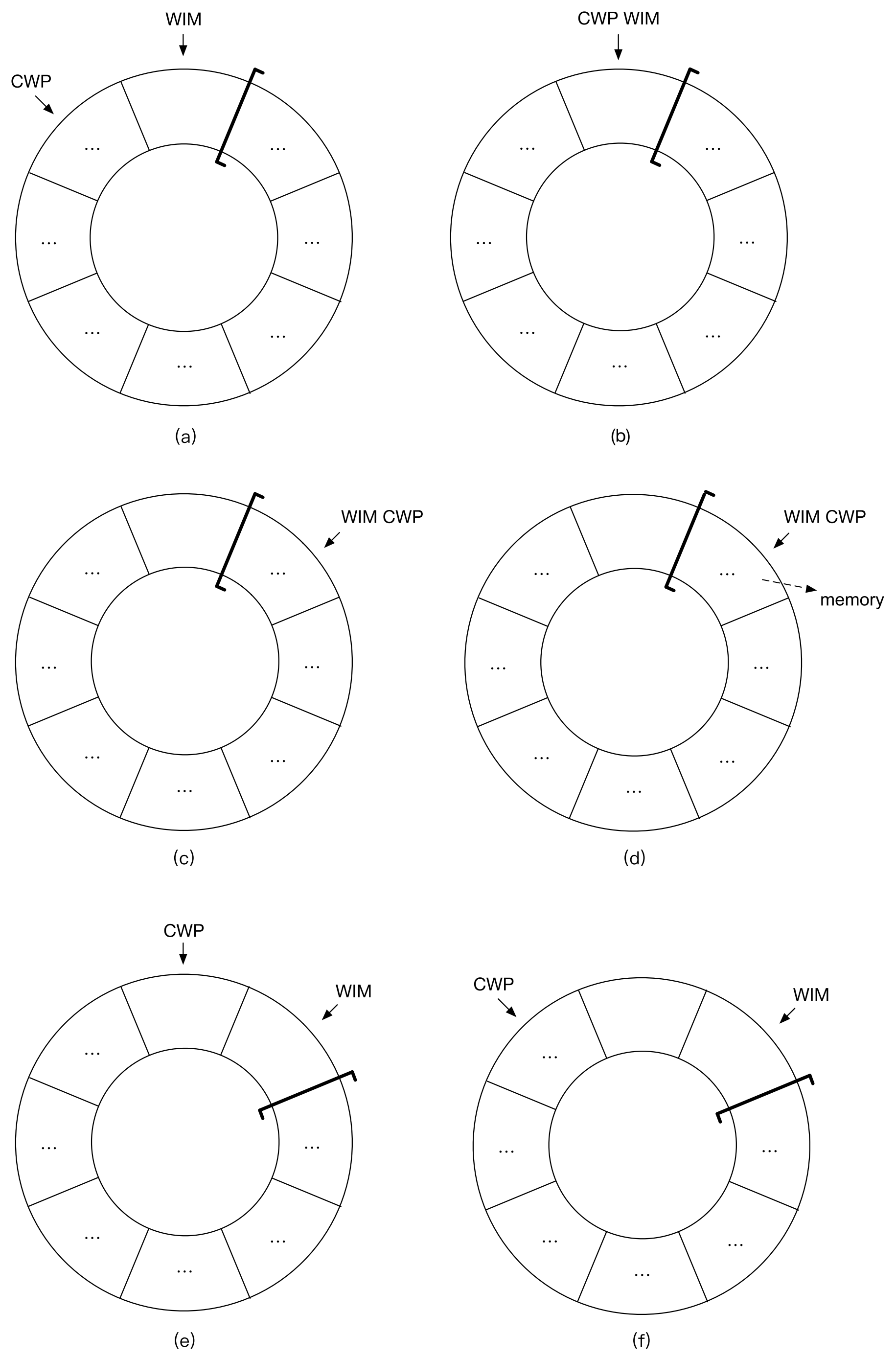

Fig. 20. Dealing with window overflow.

- Then the current window pointer cwp points to the next window, and we store the value of the current window into the memory (Lines 6 and 11-26), as shown in Fig. 20(d).
- Finally, the handler restores cwp and returns (Lines 27-30), as shown in Fig. 20(e) and Fig. 20(f).

5.3. Specifications of the window overflow handler

To verify this window overflow trap handler, we first need to give its specifications, namely the precondition and the postcondition, shown as below:

$$
\begin{array}{lll}
\text{single\_mask}(cwp, wim) & \overset{def}{=\!=} & 2^{cwp} = wim \\
\text{handler\_context}(R) & \overset{def}{=\!=} & 0 \leq cwp \leq 7 \ \wedge\ \neg\text{annulled}(R) \ \wedge\ \neg\text{trap\_enabled}(R) \ \wedge \\
 & & \neg\text{has\_trap}(R) \ \wedge\ \text{sup\_mode}(R) \\
\text{normal\_cursor}(R) & \overset{def}{=\!=} & R(npc) = R(pc) + 4 \\
\text{align\_context}(Q) & \overset{def}{=\!=} & \text{word\_aligned}(R(l_1)) \wedge \text{word\_aligned}(R(l_2)) \wedge \text{word\_aligned}(R'(sp)) \\
 & & \textbf{where}\ Q = (R, F),\ (R', F') = \text{right\_win}(R, F) \\
\text{set\_function}(a, j, C) & \overset{def}{=\!=} & \begin{cases} C(a) = i \ \wedge\ \text{set\_function}(a+4, j', C) & \textbf{if}\ j = i :: j' \\ \textbf{true} & \textbf{otherwise} \end{cases} \\
(Function)\quad j & ::= & \textbf{nil} \mid i :: j
\end{array}
$$

Fig. 21. Auxiliary definitions for pre-condition and post-condition.

$$
\begin{array}{lll}
\text{overflow\_pre\_cond}(W) & \overset{def}{=\!=} & \text{single\_invalid}(R(cwp), R(wim)) \ \wedge\ \text{handler\_context}(R) \\
 & & \wedge\ \text{normal\_cursor}(R) \ \wedge\ \text{align\_context}(Q) \ \wedge \\
 & & \text{set\_function}(R(pc),\ \texttt{WINDOW OVERFLOW},\ C_s) \ \wedge \\
 & & D = \textbf{nil} \ \wedge\ \text{length}(F) = 2N - 3 \\
 & & \textbf{where}\ W = (\Delta, (\Phi, Q, D)),\ \ \Delta = (C_u, C_s),\ \ Q = (R, F) \\
\text{overflow\_post\_cond}(W) & \overset{def}{=\!=} & \text{single\_invalid}(\text{pre\_cwp}(\text{pre\_cwp}(R)), R(wim)) \\
 & & \textbf{where}\ W = (\Delta, (\Phi, Q, D)), Q = (R, F)
\end{array}
$$

In the pre-condition, single_invalid(w, $R(wim)$) indicates that the window w is invalid, and the rest of the windows are all available. handler_context contains the unique state of the system after the exe_trap function is executed. For example, the system must be in the supervisor mode, the trap must be disabled, and so on. normal_cursor and handler_context illustrate the requirements for pc and npc before entering the overflow trap handler. align_context requires the address to be *word-aligned*. The rest gives the requirements for the delay list and the frame list. These functions mentioned above can be found in Fig. 21.

In the post-condition, when we finish running the trap handler and return to the original function where the trap occurs, cwp will point to the window used by the original function. At this point, the next window becomes valid, and the window after the next window (pre_cwp(pre_cwp(R))) is invalid.

5.4. Verify the window overflow handler

After giving the specifications of the window overflow handler, we verify the correctness of the handler by showing that the handler can be safely executed under the given pre-condition. As shown in Theorem 5, it says, if the initial state satisfies the precondition, then we can safely reach a resulting state satisfying the postcondition within 30 steps, and no trap occurs during the execution. More details about the specification and proofs can be found in the Coq implementations [9].

Theorem 5. (Correctness of the window overflow trap handler)
If overflow_pre_cond(Δ, S)*, then for all S' and E, if* $\Delta \vdash S \overset{E}{\Longmapsto}{}^{30} S'$*, then* overflow_post_cond(Δ, S') *and* no_trap_event(E).

6. Integrating SMT solvers into Coq

In low-level code, there are a lot of complex bitwise operations for improving the performance. These operations bring many challenges to verification. For example, to verify the window underflow trap handler, we have several arithmetic lemmas that need to be proved. First, it uses cyclic shift to reset the value of wim. To verify it, we need to prove Lemma 1, as shown below.

Lemma 1. (Cyclic shift)
$\forall\, n\ n'\ i\ i' \in$ *Word, if* $2^i = n$, $2^{i'} = n'$, $n' = (n \gg 1 \parallel n \ll (\texttt{NWINDOWS} - 1))$*, then* $i' = (i - 1)\ \textbf{mod}\ \texttt{NWINDOWS}$.

In Lemma 1, n and n' are one-hot encoding, and i and i' are the index of their valid bit, $n' = (n \gg 1 \parallel n \ll (\texttt{NWINDOWS} - 1))$ shows how the cyclic shift is implemented in the low level code, and $i' = (i - 1)\ \textbf{mod}\ \texttt{NWINDOWS}$ shows what we expect, that is, the index of n' is the index of n modular minus one.

And we need to prove that if the current program counter is word-aligned, the program counter will still be word-aligned after the system executes a normal step, as shown in Lemma 2.

Lemma 2. (Word-aligned forward)
$\forall\, w \in$ *Word, if* word_aligned(w)*, then* word_aligned($w + 4$).

Besides, when we verify the window underflow trap handler line by line, we also need to maintain the relations between cwp and wim. One of these relations is shown in Lemma 3.

Lemma `word_aligned_forward:`
```
    forall w, (w &i ($ 3)) = $ 0 -> ((w +i ($ 4)) &i ($ 3)) = $ 0.
```

Goal in Coq:
```
   1 subgoal
   w:int32
   H:(w &i ($ 3)) = $ 0
   ______________________________________________________________(1/1)
   ((w +i ($ 4)) &i ($ 3)) = $ 0
```

Fig. 22. The Coq format and the goal of Lemma 2.

Syntax tree in Coq:
```
H NONE
(Ind(Coq.Init.Logic.eq, 0)
Ind(SMTC.Integers.Int.int, 0)
  (Cst(SMTC.Integers.Int.and) w
    (Cst(SMTC.Integers.Int.repr)
      (Constr(Coq.Numbers.BinNums.Z, 0, 2)
        (Constr(Coq.Numbers.BinNums.positive, 0, 1)
  Constr(Coq.Numbers.BinNums.positive, 0, 3)))))
    (Cst(SMTC.Integers.Int.repr)
Constr(Coq.Numbers.BinNums.Z, 0, 1)))
w NONE Ind(SMTC.Integers.Int.int, 0)
(Ind(Coq.Init.Logic.eq, 0)
Ind(SMTC.Integers.Int.int, 0)
  (Cst(SMTC.Integers.Int.and)
    (Cst(SMTC.Integers.Int.add) w
      (Cst(SMTC.Integers.Int.repr)
        (Constr(Coq.Numbers.BinNums.Z, 0, 2)
          (Constr(Coq.Numbers.BinNums.positive, 0, 2)
            (Constr(Coq.Numbers.BinNums.positive, 0, 2)
    Constr(Coq.Numbers.BinNums.positive, 0, 3))))))
      (Cst(SMTC.Integers.Int.repr)
        (Constr(Coq.Numbers.BinNums.Z, 0, 2)
          (Constr(Coq.Numbers.BinNums.positive, 0, 1)
  Constr(Coq.Numbers.BinNums.positive, 0, 3)))))
    (Cst(SMTC.Integers.Int.repr)
Constr(Coq.Numbers.BinNums.Z, 0, 1)))
```

Syntax tree in Z3:
```

(assert (! (=

  (bvand w
    ((_ int2bv 32)
      (+
        (+ (* 2
  (^ 2 0)) 1))))
    ((_ int2bv 32)
(- (^ 2 0)1))): named H))
(declare-const w (_ BitVec 32))
(assert (! (not (=

(bvand
  (bvadd w
    ((_ int2bv 32)
      (+
        (+ (* 2
          (+ (* 2
  (^ 2 0)) 0)) 0))))
    ((_ int2bv 32)
      (+
        (+ (* 2
(^ 2 0)) 1))))
  ((_ int2bv 32)
(- (^ 2 0)1)))): named GOAL))
```

Fig. 23. Syntax tree of Lemma 2 in Coq and Z3.

Lemma 3. (Valid windows)
$\forall\, R$, *if* $0 \le R(cwp) \le \texttt{NWINDOWS} - 1$, single_mask$(R(cwp), R(wim))$, *then* ¬win_masked (pre_cwp(R), R) *and* ¬win_masked (post_cwp(R), R).

Lemma 3 says that, because the register wim is one-hot encoding, and the only invalid window is cwp indicated by the condition single_mask$(R(cwp), R(wim))$, so the previous window and the next window are all valid.

We have many other lemmas like these when verifying the low-level SPARC code. To reduce the proof burden, base on [19], we develop a tool called 'coq2smt' that can translate dozens of arithmetical operations on 8, 16, 32 and 64 bits integers from Coq into SMT solvers and solve it. Here we take the Lemma 2 as an example to show how coq2smt proves these lemmas by translating them to Z3.

Fig. 22 shows the Coq format of Lemma 2 while we unfold the definition of word_aligned and the goal when we prove it. To translate this goal into the SMT Solver Z3, first we extract the syntax tree of this goal, as shown in the left column of Fig. 23. Then we translate each element of the syntax tree into Z3 (for example, `SMTC.Integers.Int.and` to `bvand`). Finally, We prove this lemma is **true** by proving the converse proposition of this lemma is **false** (Z3 returns 'unsat' when the proposition is **false**).

7. Conclusion and future work

In this paper, we have formalized the SPARCv8 instruction set in Coq, which provides a formal model for verifying SpaceOS at the assembly level. Also the formalization can help us to add SPARCv8 into the backend of CompCert in the future. We prove the determinacy and isolation properties of the semantics to validate the model, and we also verify the window overflow handler and window underflow handler in the model.

As future work, we will model the remaining instructions, including the floating point instructions and the coprocessor instructions. To facilitate the code verification, we will develop a program logic for the assembly code, instead of doing verification based on the operational semantics directly. We hope to extend CompCert backend to support the SPARCv8 assembly language.

Also note that the correctness of the semantics model and the tool coq2smt are part of our trusted computing base. It will be important future work to further validate the semantics and the tool.

References

[1] F. Xu, M. Fu, X. Feng, X. Zhang, H. Zhang, Z. Li, A practical verification framework for preemptive os kernels, in: International Conference on Computer Aided Verification (CAV), Springer, 2016, pp. 59–79.

[2] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, et al., Sel4: formal verification of an os kernel, in: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP), ACM, 2009, pp. 207–220.

[3] The SPARC Architecture Manual Version 8 [online].

[4] L. Qiao, M. Yang, B. Gu, H. Yang, B. Liu, An embedded operating system design for the lunar exploration rover, in: Proceedings of the 2011 Fifth International Conference on Secure Software Integration and Reliability Improvement-Companion (SSIRI-C), IEEE Computer Society, 2011, pp. 160–165.

[5] X. Leroy, Formal certification of a compiler back-end, or: programming a compiler with a proof assistant, in: 33rd Symposium Principles of Programming Languages (POPL), ACM, 2006, pp. 42–54.

[6] C.A.R. Hoare, An axiomatic basis for computer programming, Commun. ACM 12 (10) (1969) 576–580.

[7] The Coq Proof Assistant [online].

[8] Count Lines of Code [online].

[9] Formalizing SPARCv8 Instruction Set Architecture in Coq (Project Code) [online].

[10] J. Wang, M. Fu, L. Qiao, X. Feng, Formalizing sparcv8 instruction set architecture in coq, in: International Symposium on Dependable Software Engineering: Theories, Tools, and Applications (SETTA), Springer, 2017, pp. 300–316.

[11] Coq to SMT Solver [online].

[12] L. De Moura, N. Bjørner, Z3: an efficient smt solver, in: International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Springer, 2008, pp. 337–340.

[13] A. Fox, M. Myreen, A trustworthy monadic formalization of the armv7 instruction set architecture, in: Interactive Theorem Proving (ITP), Springer, 2010, pp. 243–258.

[14] A. Kennedy, N. Benton, J.B. Jensen, P.-E. Dagand, Coq: the world's best macro assembler?, in: Proceedings of the 15th Symposium on Principles and Practice of Declarative Programming (PPDP), ACM, 2013, pp. 13–24.

[15] G. Gonthier, R.S. Le, An Ssreflect Tutorial, Ph.D. thesis, INRIA, 2009.

[16] Z. Hou, D. Sanan, A. Tiu, Y. Liu, K.C. Hoa, An executable formalisation of the sparcv8 instruction set architecture: a case study for the leon3 processor, in: International Conference on Formal Methods (FM), Springer, 2016, pp. 388–405.

[17] M. Armand, G. Faure, B. Grégoire, C. Keller, L. Théry, B. Werner, A modular integration of sat/smt solvers to coq through proof witnesses, in: International Conference on Certified Programs and Proofs (CPP), Springer, 2011, pp. 135–150.

[18] B. Ekici, A. Mebsout, C. Tinelli, C. Keller, G. Katz, A. Reynolds, C. Barrett, Smtcoq: a plug-in for integrating smt solvers into coq, in: International Conference on Computer Aided Verification (CAV), Springer, 2017, pp. 126–133.

[19] Invoke an SMT Solver on Coq Goals [online].

[20] X. Feng, Z. Shao, Modular verification of concurrent assembly code with dynamic thread creation and termination, in: International Conference on Functional Programming (ICFP), ACM, 2005, pp. 254–267.

[21] X. Feng, Z. Shao, Y. Dong, Y. Guo, Certifying low-level programs with hardware interrupts and preemptive threads, in: Conference on Programming Language Design and Implementation (PLDI), ACM, 2008, pp. 170–182.

[22] X. Feng, Z. Shao, A. Vaynberg, S. Xiang, Z. Ni, Modular verification of assembly code with stack-based control abstractions, in: Conference on Programming Language Design and Implementation (PLDI), ACM, 2006, pp. 401–414.

[23] A. Gaisler, S. Göteborg, Leon3 multiprocessing cpu core, Aeroflex Gaisler, February.