Formalizing SPARCv8 Instruction Set Architecture in Coq (Technical Report)

Jiawei Wang[1], Ming Fu[1], Lei Qiao[2] and Xinyu Feng[1]

[1] University of Science and Technology of China
[2] Beijing Institute of Control Engineering

Abstract. 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 system running on SPARCv8 processors, it is inevitable to consider formalizing the behavior of SPARCv8 ISA at the assembly level. In this paper, we present our formalization of the SPARCv8 ISA, which is faithful to the realistic design of SPARCv8. We also prove the deterministic and isolation properties with respect to the operational semantics of our formal model. In addition, we have verified that a trap handler function of handling window overflows satisfies the user's expectations based on our formal model. All of the formalization and proofs have been mechanized in Coq.

Keywords: SPARCv8, Coq, Formalization, Operational Semantics, Isolation, Window Overflow

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 by C language and assembly language. In existing OS verification projects, *e.g.*, CertiμC/OS-II [1], 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.*, register, stack. Therefore, the assembly code in OS kernels is not actually verified. To verify whether the assembly code satisfies its abstract specification, it is inevitable to formalize the specific behaviors of the assembly instructions.

As a high-performance, reliable, low-power microprocessors, the SPARCv8 [3] instruction set architecture (ISA) 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 China Aerospace Science and Technology Corporation (CASC) and deployed in the devices of Chang'e-3 lunar mission. On the one hand, to formally verify SpaceOS, we need to formalize the SPARCv8 instruction set and give 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 only supports translating Clight, which is an important subset of C, into ARM [6], x86 [7], PowerPC [8] instruction set currently. It does not support SPARCv8 at the backend. Extending CompCert to support SPARCv8 requires us to formalize the SPARCv8 instruction set. In this paper, we make the following contributions:

- We formalize the SPARCv8 instruction set architecture. Our formal model is faithful to the behaviors of the instructions described in the SPARCv8 manual [9], including most of the features in SPARCv8, *e.g.*, windowed register, delayed control transfer, interrupt and traps.
- We prove that the operational semantics of our formal model satisfy the determinacy property, and the execution in the user mode or the supervisor mode satisfies the isolation property.
- We take the trap handler of handling window overflows as an example, and give its pre-condition and post-condition to specify the expected behaviors. Like proving programs with Hoare triples, we prove that the trap handler satisfies the given pre-/post-conditions and does not throw any exceptions.
- All of the formalization and proofs have been mechanized in Coq [10]. They contain around 11000 lines of coq scripts in total. The source code can be accessed via the link [19].

Related Work. Fox and Myreen gave the ARMv7 ISA model [11], 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 [12], and they used type classes, notations and the mathematics library Ssreflect [13]. 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 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 tool Isabelle [14], which is closed 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 operation 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 uncertainty of the operational semantics caused by interrupt. Besides, our formalization of the

SPARCv8 ISA is implemented in Coq, while CompCert is implemented in Coq too. We are able to use our Coq implementation to extend the CompCert at the backend to support SPARCv8 in the future.

There are some other verification work at assembly level [15, 16, 17], which give the formal model of the subset of x86 the instruction set and the behavior of the x86 interrupt management. They mainly study the verification technology of x86 assembly code, the instruction set is relatively small. In the meanwhile, the model is simple. We formalized the SPARCV8 ISA by considering all the features of SPARCv8. In the next section, we will 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 system of space device. For example, LEON3 [18], a SPARCv8 architecture-based processor, developed by the European Space Research and Technology Center, is widely used in ASICs.

Compared to with x86 and ARM, SPARCv8 has better performance, lower power consumption and higher reliability. These specialties are due to the following unique mechanisms:

– A variety of control-transfer instructions (CTIs) and annulled delay instructions for more flexible function jumps.
– Register windows 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 several cycles.

These characteristics pose quite a few challenges for formal modeling. We will use an 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 `SUM3` requires three input parameters. When the `CALLER` calls `SUM3`, it places the first two arguments, then calls `SUM3` (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. The real transfer procedure is executed in the next instruction cycle. This feature is called "delayed transfer", which also happens at lines 9 and 10.

In `SUM3`, we use `save` and `restore` instructions (Lines 6 and 10) to save and restore the caller's context. When this program is running, both `CALLER` and `SUM3` will have a register window as there context, and their windows are overlapping. When the `CALLER` needs to save the context and pass the parameters to `SUM3`, it will put the parameters in the overlapping section and rotates the window so that the `SUM3`'s register window is exposed. At this point, the non-overlapping portion of the `CALLER`'s window is hidden. These steps are implemented by the `save` instruction. When `SUM3` needs to pass the return value to the `CALLER`, it will put the return value in the overlap section and rotate the window to destroy its own space. 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 formalize the SPARCv8 ISA and verify the correctness of the SPARCv8 code.

3 Modeling SPARCv8 ISA

The SPARCv8 instruction set provides programmers with a hardware-oriented assembly programming language. To formalize it, first we need to provide the abstract syntax of the given language. Then we define the machine state. Finally, we give the operational semantics for the instructions. We present these three parts in section 3.1, section 3.2 and section 3.3, respectively.

3.1 Syntax

Fig. 1 shows the syntax of the SPARCv8 assembly language.

$$
\begin{array}{rcll rcll}
(\mathit{Word}) & w & \in & \mathit{Int}32 & (\mathit{OpExp}) & \alpha & ::= & r \mid w \\
(\mathit{GenReg}) & r & ::= & r_0 \mid \ldots \mid r_{31} & (\mathit{AddrExp}) & \beta & ::= & \alpha \mid r + \alpha \\
(\mathit{AsReg}) & asr & ::= & asr_0 \mid \ldots \mid asr_{31} & (\mathit{TrapExp}) & \gamma & ::= & r \mid r + r \mid r + w \mid w \\
(\mathit{Symbol}) & \varsigma & ::= & psr \mid wim \mid tbr \mid y \mid asr & (\mathit{TestCond}) & \eta & ::= & al \mid nv \mid ne \mid eq \mid \ldots \\
(\mathit{SparcIns}) & i & ::= & \multicolumn{5}{l}{\textbf{ticc}\ \eta\ \gamma \mid \textbf{rett}\ \beta \mid \textbf{save}\ r_s\ \alpha\ r_d \mid \textbf{restore}\ r_s\ \alpha\ r_d \mid \textbf{ld}\ \beta\ r_d} \\
 & & & \multicolumn{5}{l}{\mid \textbf{st}\ r_d\ \beta \mid \textbf{rd}\ \varsigma\ r_d \mid \textbf{wr}\ r_d\ \alpha\ \varsigma \mid \textbf{bicc}\ \eta\ \beta \mid \textbf{jmpl}\ \beta\ r_d \mid \textbf{nop} \mid \ldots}
\end{array}
$$

Fig. 1. The syntax of the SPARCv8 assembly language

w stands for 32-bit integers (*Word*). *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}. We use ς to represent symbol registers (*Symbol*), including the processor state register (*psr*), window invalid mask register (*wim*), trap base register (*tbr*), multiply/divide register (y) and ancillary state registers (*asr*). Symbol registers can only be read and written by **rd** and **wr** instructions.

Table 1. General registers and corresponding aliases

General Register	Alias
r0 - r7	g0 - g7
r8 - r15	o0 - o7
r16 - r23	l0 - l7
r24 - r31	i0 - i7
r14	sp
r30	fp

Note 3.1. sp refers to the stack pointer, fp refers to the frame pointer

r stands for general registers(*GenReg*), with names ranging from r_0 to r_{31}. These registers also have some aliases, as shown in Tab. 1. Note that the r0 (g0) register is a special register, Its value is constant to 0. So the read operation on this register will always return 0, and the write operation does noting, as shown bellow.

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

We use α, β and γ to represent operand expressions (*OpExp*), address expressions (*AddrExp*) and trap expressions (*TrapExp*) respectively. 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 bellow.

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

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

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

When these expressions are evaluated, if the required range of these values is not satisfied, the function will return `NONE`.

η represents the conditional expression, containing always(al), never(nv), not equal(ne), equal(eq) conditional expressions and so on. When evaluating conditional expressions, we need to determine whether the given condition is hold by reading the conditional registers (n, z, v, c), as shown bellow.

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

i is used to represent the SPARCv8 assembly instructions ($SparcIns$). Here we only give some typical instructions, as shown in Fig 1, the omitted instructions are all arithmetic instructions. **ticc** is used to trigger software traps. **rett** is used to return from traps. **save** and **restore** are used to save and restore the parental function's context. **ld** and **st** are used to load values from memory and store values into memory. **rd** and **wr** are used to read values from symbol registers and write values into symbol registers. **bicc** and **jmpl** are used to jump to some specified locations. **nop** does nothing. The conditional expression and **ticc** or **bicc** are synthesized into a SPARC assembly instruction For example, **ticc** al instruction represents the **ta** instruction in the original syntax, and **ticc** nv represents the **tb** instruction. The **bicca** instruction represents an **bicc** instruction with annulled feature. For example, **bicca** al instruction represents the **ba, a** instruction in the original syntax.

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 obtained from these basic instructions in Fig. 1. The synthetic method can be found in the SPARCv8 manual [9].

3.2 Machine States

We first define the machine state for the operational semantics.

Register File. Here we give the definition of the register file ($RegFile$) shown as bellow.

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

We use μ and ν to represent the processor state register ($PsrSeg$) and the trap base register ($TbrSeg$) respectively. These two registers are all 32 bits, and

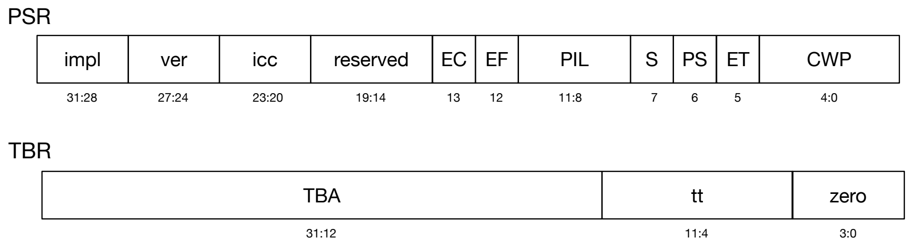

Fig. 2. PSR and TBR segment

they contain several segments to represent the different status of the processor, as shown in Fig. 2. The ET segment determines whether traps are enabled. The S segment determines whether the processor is in supervisor or user mode. The PS segment contains the value of the S segment at the time of the most recent trap. The TBA segment represents the trap base address, which is established by supervisor software. We split PSR and TBR into several parts for simplifying the formalization of the machine state. After splitting, we can define some predicates on these segments, as shown bellow.

$$\mathsf{trap\_disabled}(R) \overset{def}{=\!=} R(et) = 0 \qquad \mathsf{usr\_mode}(R) \overset{def}{=\!=} R(s) = 0$$
$$\mathsf{trap\_enabled}(R) \overset{def}{=\!=} R(et) \neq 0 \qquad \mathsf{sup\_mode}(R) \overset{def}{=\!=} R(s) \neq 0$$

If the value of the register et is 0, the trap is disabled, and if the value of the register et is not 0, the trap is enabled. The machine is in the user mode when the value of the register s is 0, and it is in the supervisor mode when the value of the register s is not 0.

Besides these predicates, we can also define some functions on these registers, such as switching mode between user mode and supervisor mode ($\mathsf{to\_usr}$, $\mathsf{to\_sup}$), save mode($\mathsf{save\_mode}$), save the value of the register s into the register ps, enable or disable the trap ($\mathsf{enable\_trap}$, $\mathsf{disable\_trap}$) and so on, as shown bellow.

$$\mathsf{to\_usr}(R) \overset{def}{=\!=} R\{s \rightsquigarrow 0\} \qquad \mathsf{to\_sup}(R) \overset{def}{=\!=} R\{s \rightsquigarrow 1\}$$
$$\mathsf{save\_mode}(R) \overset{def}{=\!=} R\{ps \rightsquigarrow R(s)\} \qquad \mathsf{restore\_mode}(R) \overset{def}{=\!=} R\{s \rightsquigarrow R(ps)\}$$
$$\mathsf{enable\_trap}(R) \overset{def}{=\!=} R\{et \rightsquigarrow 1\} \qquad \mathsf{disable\_trap}(R) \overset{def}{=\!=} R\{et \rightsquigarrow 0\}$$

Since we split PSR and TBR into several parts, we need to define the read operations of these registers, as shown bellow.

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

We use q to represent the register name ($RegName$), including r, wim, y, asr, psr and tbr, which are explained before. It also includes the program counter pc, the next program counter npc, the register τ that records trap states and the register κ for annulled states. Annulled states can be triggered by the annulled delay instructions. If a system is in the annulled state, the next instruction will be ignored (assuming a trap does not occur). A register file R is modeled as a total function mapping register names to 32-bit integers. Here we give some predicates and functions on the register τ and κ, as shown bellow.

$$\begin{array}{llllll}
\mathsf{annuled}(R) & \stackrel{def}{=\!=} & R(\kappa) \neq 0 & \mathsf{has\_trap}(R) & \stackrel{def}{=\!=} & R(\tau) \neq 0 \\
\mathsf{set\_annul}(R) & \stackrel{def}{=\!=} & R\{\kappa \rightsquigarrow 1\} & \mathsf{set\_trap}(R) & \stackrel{def}{=\!=} & R\{\tau \rightsquigarrow 1\} \\
\mathsf{clear\_annul}(R) & \stackrel{def}{=\!=} & R\{\kappa \rightsquigarrow 0\} & \mathsf{clear\_trap}(R) & \stackrel{def}{=\!=} & R\{\tau \rightsquigarrow 0\}
\end{array}$$

The machine is in the trap state if the value of the register τ is not 0, and it is in the annulled state if the value of the register κ is not 0. We also have some functions to set and clear the annulled state (set_annul, clear_annul) or to set and clear the trap state (set_trap, clear_trap).

In SPARCv8 ISA, it uses two program counters, *e.g.*, 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 to be executed (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:

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

For non-jump instructions, we use next to update pc with npc and increases npc by 4. For jump instructions, we use djmp to update pc with npc and set npc to the target address. 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".

Frame List. We use the frame and the frame list to describe the feature of window registers and window rotating. The definition of them are given as follows:

$$\begin{array}{llll}
(Frame) \; f & ::= & [w_0, \ldots, w_7] & (FrameList) \; F \; ::= \; \mathbf{nil} \mid f :: F \\
(RState) \; Q & ::= & (R, F) &
\end{array}$$

A frame is an array that contains 8 words, and a frame list is a list of frames and its length is 2N-3 (N is the number of windows). At a given time, an instruction can access the 8 globals and a 24-register window into the general registers. The general registers have been modeled as the register file R. So next we need to model the registers that can't be accessed by instructions. We use the frame list to record the values in these registers.

We classify registers according to whether they can be accessed, but whether registers can be accessed is not changeless. This classification may change when the window is rotated, as what the `caller` and `sum3` do in the example in Sec. 2. We pair the register file and the frame list together as the register state Q to facilitate the definition of window rotation operations. There are three window rotation operations on Q, as shown bellow.

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

$$\mathsf{inc\_win}(Q) \overset{def}{=\!=} \begin{cases} \mathsf{left\_win}(1, Q) & \textbf{if } \neg\mathsf{win\_masked}(\mathsf{post\_cwp}(1, R), R) \\ \bot & \textbf{otherwise} \end{cases}$$
$$\textbf{where } Q = (R, F)$$

$$\mathsf{dec\_win}(Q) \overset{def}{=\!=} \begin{cases} \mathsf{right\_win}(1, Q) & \textbf{if } \neg\mathsf{win\_masked}(\mathsf{pre\_cwp}(1, R), R) \\ \bot & \textbf{otherwise} \end{cases}$$
$$\textbf{where } Q = (R, F)$$

where

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

$\mathsf{set\_win}(w, Q)$ operation takes the window w as the current window, $\mathsf{inc\_win}$ and $\mathsf{dec\_win}$ operations are used to increase and decrease the label of the current window respectively. $\mathsf{inc\_win}$ and $\mathsf{dec\_win}$ first need to check whether the window to be used is masked by using function $\mathsf{win\_masked}$. Then they rotate the window to left or right 1 time.

When the window needs to rotate to the left 1 time, the following operations will be required:

– As shown in Fig.3(1), the out, local and in are converted into 3 frames:

$$R[r_i, \ldots, r_{i+7}] \overset{def}{=\!=} [R(r_i), \ldots, R(r_{i+7})]$$
$$R\{[r_i, \ldots, r_{i+7}] \rightsquigarrow f\} \overset{def}{=\!=} R\{r_i \rightsquigarrow w_0\} \ldots \{r_{i+7} \rightsquigarrow w_7\}$$
$$\textbf{where } f = [w_0, \ldots, w_7]$$
$$\mathsf{fetch}(R) \overset{def}{=\!=} R[r_8, \ldots, r_{15}] :: R[r_{16}, \ldots, r_{23}] :: R[r_{24}, \ldots, r_{31}] :: \textbf{nil}$$

– As shown in Fig. 3(2) and Fig. 3(4), we can insert these 3 frames at the end of the frame list, then rotate the frame list.

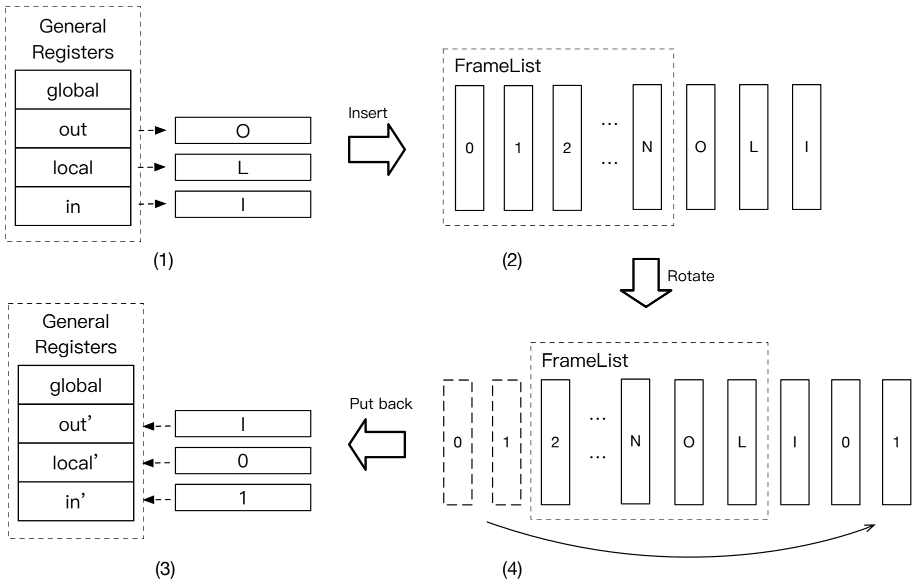

Fig. 3. Left rotation of the window

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

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

– Finally, as shown in Fig. 3(3) and Fig. 3(4), we remove 3 frames from the tail of the frame list, then insert them to the corresponding positions in the 32 general registers.

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

These steps rotate the window to the left once, as shown in Fig.3. By combining the above three steps, we can get the left rotation of the window:

$$\mathsf{left\_win}(Q) \overset{def}{=\!=} \mathbf{let}\ (F', l) ::= \mathsf{left}(F, \mathsf{fetch}(R))\ \mathbf{in}$$
$$\mathbf{let}\ R' ::= \mathsf{replace}(l, R)\ \mathbf{in}\ (R'\{cwp \rightsquigarrow \mathsf{post\_cwp}(R)\}, F')$$
$$\mathbf{where}\ Q = (R, F),\ \mathsf{post\_cwp}(R) \overset{def}{=\!=} (R(cwp) + 1)\ \mathbf{mod}\ N$$

The right rotation of the window is similar to left rotation, as shown bellow.

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

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

$$\begin{aligned}
\mathsf{right\_win}(w, Q) &\overset{def}{=\!=} \mathbf{let}\ (F', l) ::= \mathsf{right\_iter}(w, F, \mathsf{fetch}(R))\ \mathbf{in} \\
&\qquad \mathbf{let}\ R' ::= \mathsf{replace}(l, R)\ \mathbf{in}\ (R'\{cwp \rightsquigarrow \mathsf{pre\_cwp}(w, R)\}, F') \\
&\quad \mathbf{where}\ Q = (R, F),\ \mathsf{pre\_cwp}(w, R) \overset{def}{=\!=} (R(cwp) - w + N)\ \mathbf{mod}\ N
\end{aligned}$$

We also have some predicates defined on register state, as shown bellow.

$$\begin{aligned}
\mathsf{has\_trap}(Q) &\overset{def}{=\!=} \mathsf{has\_trap}(R) && \mathbf{where}\ Q = (R, F) \\
\mathsf{usr\_mode}(Q) &\overset{def}{=\!=} \mathsf{usr\_mode}(R) && \mathbf{where}\ Q = (R, F) \\
\mathsf{sup\_mode}(Q) &\overset{def}{=\!=} \mathsf{sup\_mode}(R) && \mathbf{where}\ Q = (R, F) \\
\mathsf{annuled}(Q) &\overset{def}{=\!=} \mathsf{annuled}(R) && \mathbf{where}\ Q = (R, F)
\end{aligned}$$

Delay List. In SPARCv8, when we execute the **wr** instruction to write the symbol register, the execution will be delayed for several cycles. And the delayed cycle is implementation-dependent. In order to describe the delayed write feature, we use the delay list defined as below to store the data that related to the delayed write operation.

$$\begin{aligned}
&(\mathit{DelayCycle})\ \ c\ \ \in\ \ \{0, 1, \ldots, X\} \qquad (\mathit{DelayItem})\ \ d\ ::=\ (c, \varsigma, w) \\
&(\mathit{DelayList})\ \ D\ ::=\ \mathbf{nil} \mid d :: D
\end{aligned}$$

The item of the delay list is a triple consisting of the delayed cycle (*DelayCycle*), the register name, and the value that needs to be written in the register. When we execute the **wr** instruction, we will insert a triple with delayed information into the delay list. Besides the insertion operation, we also need to check the delay list at the beginning of each instruction cycle. If the delay cycle of an item in it is 0, we will remove this item from the delay list and update the corresponding register with the value in the triple. If the delay cycle of an item in it is not 0, it will reduce the delay cycle by 1. The detail of these operations can be found in the Sec. 3.3.

Machine States and Code Heap. We use M to represent the memory (*Memory*), which maps the addresses (*Address*) to words. The full memory is split into two parts for the user mode and the supervisor mode respectively. We formalize the memory as a pair that consists of the user memory M_u and the supervisor memory M_s. The machine state S contains the memory pair Φ, the register state Q and the delay list D.

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

We can define some predicates on register state, as shown bellow.

$$\begin{array}{lcll}
\mathsf{has\_trap}(S) & \overset{def}{=\!=} & \mathsf{has\_trap}(Q) & \textbf{where } S = (\Phi, Q) \\
\mathsf{usr\_mode}(S) & \overset{def}{=\!=} & \mathsf{usr\_mode}(Q) & \textbf{where } S = (\Phi, Q) \\
\mathsf{sup\_mode}(S) & \overset{def}{=\!=} & \mathsf{sup\_mode}(Q) & \textbf{where } S = (\Phi, Q)
\end{array}$$

Besides the machine state, we also define the code heap C , the pair of code heap Δ and the event e, shown as bellow.

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

C represents the code heap, which maps the labels to the instructions. The code heap of user mode and supervisor mode together form the pair of code heap Δ. The whole world W consists of the code heap Δ of two modes and the machine state S.

e stands for events. If a trap occurs, the corresponding trap label w is recorded as an event, otherwise it is $\bot$. An event list E is introduced for producing events of the multi-step execution.

3.3 Operational Semantics

$$\begin{tikzcd}
\boxed{(M, R) \xrightarrow{i} (M', R')} \arrow[d, Rightarrow] & \text{Simple Instructions} \\
\boxed{(M, Q, D) \circ\!\xrightarrow{i} (M', Q', D')} \arrow[d, Rightarrow] & \text{Windowed Register and Delayed Write} \\
\boxed{C \vdash (M, Q, D) \bullet\!\longrightarrow (M', Q', D')} \arrow[d, Rightarrow] & \text{Annulled State} \\
\boxed{\Delta \vdash S \overset{e}{\Longrightarrow} S'} & \text{Interrupt, Traps and Mode Switch}
\end{tikzcd}$$

Fig. 4. The architecture of operational semantics

We define the operational semantics with multiple layers as shown in Fig. 4, where the main features of SPARCv8 are introduced at different layers. This layered operational semantics is good for our verification work, for example, when we verify arithmetic instructions such as **jmpl**, **ld**, **st** and so on, 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 the manual or [14] does, all the registers will always show up in the verification process.

In Fig. 4, from the top to the bottom, we first define the operational semantics of some simple instructions which only access the register file and memory using the transition $(M, R) \xrightarrow{i} (M', R')$.

Secondly, we lift the first layer and give the operational semantic of specific instructions about the window register and delayed write features using the transition $(M, Q, D) \circ\!\xrightarrow{i} (M', Q', D')$.

Thirdly, we use the transition $C \vdash (M, Q, D) \bullet\!\longrightarrow (M', Q', D')$ to define the operational semantic of annulled state and delayed write.

Finally, we give the operational semantic rules of interrupt, trap execution and mode switch as the transition $\Delta \vdash S \xRightarrow{e} S'$, which defines the whole behavior of the entire program. Next, we will introduce some rules of the operation semantics of each layer.

Simple Instructions. Here we give some rule of simple instructions.

For the rule JMPL shown as below, we first evaluate the address expression β ($[\![\, \beta \,]\!]_R$) to get the value w and require the address w to be word aligned using the predicate $\mathsf{word\_aligned}$. Then we save the value of pc into register r_d with the function $\mathsf{save\_pc}$. Finally, we jump to the target address w in terms of the delayed-transfer function djmp defined in Sec. 3.2.

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

where

$$\mathsf{word\_aligned}(w) \overset{def}{=\!=} w_{<1:0>} = 0, \ \mathsf{save\_pc}(r_i, R) \overset{def}{=\!=} R\{r_i \rightsquigarrow R(pc)\}$$

The rule NOP requires noting.

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

For the rule LD, we first evaluate the address expression β to get the value w which should be word aligned and in the domain of M. Then we store the value $M(w)$ into the register r_d. Finally, we move both pc and npc with the function next defined in Sec. 3.2.

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

For the rule ST, we first evaluate the address expression β to get the value w which should be word aligned and in the domain of M. Then we update the memory by storing the address w with the value of r_d. Finally, we move both pc and npc with the function next defined in Sec. 3.2.

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

For the rules BICC-TRUE and BICC-FALSE, we first evaluate the address expression β to get the value w and require the address w to be word aligned. If the value of the conditional expression η is true, we execute the delayed transfer by using function djmp, otherwise we only execute the function next.

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

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

For the rules BICCA-FALSE, BICCA-TRUE and BICCA-ALWAYS, we first evaluate the address expression β to get the value w and require the address w to be word aligned. Then we choose whether to transfer by the value of test condition.

– If the value of the test condition is false, we only set the system to annulled state.

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

– If the type of the test condition is al, we only execute the delayed transfer.

$$\frac{[\![\, \beta \,]\!]_R = w \qquad \mathsf{word\_aligned}(w)}{(M, R) \xrightarrow{\mathbf{bicca}\ al\ \beta} (M, \mathsf{set\_annul}(\mathsf{djmp}(w, R)))} \quad (\textsc{bicca-always})$$

– If the value of the test condition is type, we execute the delayed transfer and set the system to annulled state.

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

For the rules TICC-TRUE and TICC-FALSE, we first evaluate the address expression γ and require the value of it is not $\bot$. If the value of the conditional expression η is true, we set the software trap by using function set_user_trap, otherwise we do nothing.

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

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

where

$$\mathsf{set\_user\_trap}(k, R) \overset{def}{=\!=} \mathsf{set\_trap}(R\{tt \rightsquigarrow 128 + k\})$$

Rules RD-USR and RD-SUP correspond to user mode and supervisor mode respectively. The wr instruction can only read the y register or asr register when the system is in user mode. If it satisfies the read permission above, it will put the value of ς into the register r_d.

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

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

For the rule UDIVCC, we first evaluate the operand expression α to get the value w and require it is not 0. Then we take the value of y and the value of r_s as the dividend and take the value of r_d as the divisor to perform the division operation. According to the state of result, such as negative (n), zero (z), overflow(v) and carry (c), we modify the corresponding registers. As shown bellow.

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

where

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

$$\mathsf{set\_icc}(k, R) \overset{def}{=\!=} R\{n \rightsquigarrow w_n\}\{z \rightsquigarrow w_z\}\{v \rightsquigarrow w_v\}\{c \rightsquigarrow w_c\}$$
$$\textbf{where } k = (w_n, w_z, w_v, w_c)$$

Window Register and Delayed Write. Here we give some instructions with respect to the frame list and the delay list.

For the rule LIFT1, we lift the transition $(M, R) \xrightarrow{i} (M', R')$ to the transition $(M, Q, D) \circ\!\!\xrightarrow{i} (M', Q', D')$.

$$\frac{(M, R) \xrightarrow{i} (M', R')}{(M, (R, F), D) \circ\!\!\xrightarrow{i} (M', (R', F), D)} \quad (\text{LIFT1})$$

For the rule SAVE, we first decrease the label of the window using the function $\mathsf{dec\_win}$. Then we evaluate the operand expression α to get the value a. Next we assign the value of $a{+}r_s$ to r_d. Finally, we move both pc and npc with the function next defined in Sec. 3.2.

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

For the rule RESTORE, we first increase the label of the window with the function $\mathsf{inc\_win}$. Then we evaluate the operand expression α to get the value a. Next we assign the value of $a{+}r_s$ to r_d. Finally, we move both pc and npc with the function next defined in Sec. 3.2.

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

For the rule WR-USR, WR-SUP and WR-PSR, we fist evaluate the operand expression α to get the value a. Then we execute the XOR operation of a and r_s to get the value w. Next we insert the three tuples of the initial delayed cycle X, the register name ς, and the value w into the delay list. Finally, we move both pc and npc with the function next defined in Sec. 3.2. Note that when we write the PSR register, the ET and PIL should to be written immediately, with respect to interrupts.

$$\frac{\begin{array}{l}\mathsf{usr\_mode}(R) \qquad \varsigma = y \ \textbf{or}\ asr_i \qquad [\![\,\alpha\,]\!]_R = a \\ [\![\,r_s\,]\!]_R\ \textbf{xor}\ a = w \qquad D' = \mathsf{set\_delay}(\varsigma, w, D)\end{array}}{(M,(R,F),D) \xhookrightarrow{\textbf{wr}\ r_d\ \alpha\ \varsigma} (M,(\mathsf{next}(R),F),D')} \quad (\text{WR-USR})$$

$$\frac{\begin{array}{l}\mathsf{sup\_mode}(R) \qquad \varsigma \neq psr \qquad [\![\,\alpha\,]\!]_R = a \\ [\![\,r_s\,]\!]_R\ \textbf{xor}\ a = w \qquad D' = \mathsf{set\_delay}(\varsigma, w, D)\end{array}}{(M,(R,F),D) \xhookrightarrow{\textbf{wr}\ r_d\ \alpha\ \varsigma} (M,(\mathsf{next}(R),F),D')} \quad (\text{WR-SUP})$$

$$\frac{\begin{array}{l}\mathsf{sup\_mode}(R) \qquad [\![\,\alpha\,]\!]_R = a \qquad [\![\,r_s\,]\!]_R\ \textbf{xor}\ a = w \qquad w_{<4:0>} < N \\ D' = \mathsf{set\_delay}(psr, w, D) \qquad R' = R\{et \rightsquigarrow w_{<5>}\}\{pil \rightsquigarrow w_{<11:8>}\}\end{array}}{(M,(R,F),D) \xhookrightarrow{\textbf{wr}\ r_s\ \alpha\ psr} (M,(\mathsf{next}(R'),F),D')} \quad (\text{WR-PSR})$$

where

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

Exceptions. The above rules show some operational semantics of the assembly instructions. Most of the conditions of these rules are used to prevent the exceptions. If these conditions are not satisfied, the system will throw exceptions. Exceptions include traps and abortions.

Traps such as divided by zero, memory not aligned, window overflow, and so on, will not make the machine to get stuck, but will put the trap type label into the trap type register (tt), then the system will execute this trap in the next cycle. As shown bellow.

$$\begin{array}{llllll}
illegal\_ins & \stackrel{def}{=\!=} 2 & mem\_not\_align & \stackrel{def}{=\!=} 7 & win\_overflow & \stackrel{def}{=\!=} 5 \\
privileged\_ins & \stackrel{def}{=\!=} 3 & div\_by\_zero & \stackrel{def}{=\!=} 42 & win\_underflow & \stackrel{def}{=\!=} 6
\end{array}$$

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

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

As shown above, **trap_type** function first judge whether or not there is a trap generated by instructions. If the instruction causes a trap, it will return the trap type. In details:

– It will cause a *privileged_instruction* trap if the **rd** or **wr** instruction attempt to access the *wim*, *psr*, and *tbr* registers in user mode, or the **rett** instruction is executed when the system is in user mode and the trap is enabled.

- An *illegal_ins* trap is generated if the **rett** instruction is executed when the system is in supervisor mode and the trap is enabled, or the new value of *cwp* that given by instruction **wr** is out of range.
- The *win_overflow* or *win_underflow* trap is occurred if the window to be used is masked when executing **save** or **restore** instruction.
- An *mem_not_align* trap is occurred if the address in instruction **ld**, **st**, **jmpl** or **bicc** is not memory aligned.
- An *div_by_zero* trap is generated if the devisor in instruction **udivcc** is 0.

The function unexpected_trap first calls the trap_type function to determine whether the instruction has a trap, and if that happens, writes the trap type to the register *tt* and set the system to the trap state. Otherwise, it returns $\perp$. Note that unexpected_trap only detects the trap that makes an exception, and ignores the trap generated by the user through **ticc** instructions.

The abortions make the machine to get stuck.

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

As shown above. Abortion instructions are divided into 3 categories:

- If the immediate value of α, β and γ expressions do not satisfy the range conditions.
- When **ld** and **st** instructions attempt to access the memory, the address given by instructions is not within the domain of memory.
- When executing **rett** to return from the trap, the system is in user mode or the address given by the instruction is not word aligned, or the next window is masked.

The rules for unexpected traps and abortions are as follows.

$$\frac{\mathsf{unexpected\_trap}(i, Q) = Q'}{(M, Q, D) \overset{i}{\circ\!\!\longrightarrow} (M, Q', D)} \qquad \frac{\mathsf{abort\_ins}(i, Q, M)}{(M, Q, D) \overset{i}{\circ\!\!\longrightarrow} \mathbf{abort}}$$

Annulled State. Here we deal with the delayed state transition and the annulled state transition.

– For the following rule, the exe_delay function checks whether the delay list has items with a delay cycle of 0. If the system is not in the annulled state (annulled), it will pick up an instruction from the code heap and execute it.

$$\frac{\mathsf{exe\_delay}(Q, D) = (Q', D') \qquad \neg\mathsf{annulled}(Q') \qquad C(\mathsf{cursor}(Q')) = i \qquad (M, Q', D) \overset{i}{\circ\!\!\longrightarrow} (M, Q'', D'')}{C \vdash (M, Q, D) \bullet\!\!\longrightarrow (M, Q'', D'')}$$

The exe_delay function is defined as follow.

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

where

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

If the delay cycle of an item in it is 0, it will remove this item from the delay list and update the corresponding register with the value in the triple. If the delay cycle of an item in it is not 0, it will reduce the delay cycle by 1. For the *y*, *asr* register, the whole registers need to be written. For the *wim* register, we just need to write the bits that related to the windows of the system, namely 0 to N-1 bits. For the *psr* register, we have written the ET and PIL segments before, so we ignore these segments here.

– For the following rule, the $\mathsf{exe\_delay}$ function checks the delay list. If the system is in the annulled state ($\mathsf{annulled}$), it will skip one instruction and clear the annulled state ($\mathsf{clear\_annul}$).

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

where

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

Besides these 2 rules, if the instruction cannot be fetched from the code heap, or if the instruction is executed abnormally, the machine is aborted. As shown bellow.

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

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

Interrupt, Traps and Mode Switch. At the beginning of the instruction cycle, we deal with the interrupt request first. The interrupt request is also one of the traps and has the lowest priority. The execution of the trap always takes the highest priority trap for execution, and interrupts can only produce one at a time. Therefore, if a trap has already occurred in the machine, it must not be interrupted. If there is an interruption request, the machine needs to check whether a trap is allowed and whether the interrupt level meets the requirements, as shown bellow.

$$\mathsf{interrupt}(w, Q) \overset{def}{=\!=} \begin{cases} \mathsf{set\_trap}(R\{tt \rightsquigarrow 16 + w\}) & \textbf{if } \neg\mathsf{has\_trap}(R), \mathsf{trap\_enabled}(R), \\ & \;\; 1 \le w \le 15, w = 15 \;\vee\; R(pil) < w \\ \bot & \textbf{otherwise} \end{cases}$$
$$\textbf{where } Q = (R, F)$$

The interrupt function is to determine whether the interrupt is allowed. The input parameters w is the interrupt request level. If the interrupt request satisfies the condition, the trap type is recorded in the trap type register tt and the system is in a trap state. Otherwise the function return $\bot$. If the interrupt request is allowed, it will be handled by the trap execution function like the other traps. The trap execution function is shown bellow.

$$\mathsf{exe\_trap}(Q) \overset{def}{=\!=} \begin{cases} \textbf{let } (R', F') ::= \mathsf{right\_win}(1, Q) \;\; \textbf{in} & \\ \textbf{let } R'' ::= \mathsf{to\_sup}(\mathsf{save\_mode}(\mathsf{disable\_trap}(R'))) \;\; \textbf{in} & \\ (\mathsf{clear\_trap}(\mathsf{save\_pc\_npc}(r_{17}, r_{18}, R'')), F') & \textbf{if } \mathsf{trap\_enabled}(R) \\ \bot & \textbf{otherwise} \end{cases}$$
$$\textbf{where } Q = (R, F)$$

The $\mathsf{exe\_trap}$ function first checks whether the trap is disabled. If it is true, the null value is returned directly. Otherwise, we first rotate the window to right 1 times and disable trap, then save current code to register ps and turn into the supervisor mode, and finally save the value of register pc and npc to register r_{17}, r_{18} and clear the trap state. When we save the values of PC and nPC, we also need to consider the invalid instruction state, as shown bellow.

$$\mathsf{save\_pc\_npc}(r_m, r_n, R) \overset{def}{=\!=\!=} \begin{cases} R\{r_m \rightsquigarrow R(pc)\}\{r_n \rightsquigarrow R(npc)\} & \textbf{if } \neg\mathsf{annuled}(R) \\ \mathsf{clear\_annul}(R\{r_m \rightsquigarrow R(npc)\} \\ \qquad\qquad \{r_n \rightsquigarrow R(npc+4)\}) & \textbf{otherwise} \end{cases}$$

If the current system is in an annulled state, we actually save the value of npc and $npc + 4$, because the instruction at address pc is annulled.

The rules for interrupt, trap execution and mode switch are given below.

– For the following rule, if there is an interrupt request and the request is allowed (the return value of function $\mathsf{interrupt}$ isn't $\perp$), the system triggers a trap after this external interrupt happens. It will record the trap type ($\mathsf{get\_tt}$) and execute this trap ($\mathsf{exe\_trap}$) to response the interrupt request, then it will dispatch an instruction.

$$\frac{\mathsf{interrupt}(w, Q) = Q' \qquad \mathsf{get\_tt}(Q') = w \qquad \mathsf{exe\_trap}(Q') = Q'' \\ C_s \vdash (M_s, Q'', D) \bullet\!\!\longrightarrow (M_s', Q''', D')}{(C_u, C_s) \vdash ((M_u, M_s), Q, D) \xRightarrow{w} ((M_u, M_s'), Q''', D')}$$

– For the following rule, if the machine has traps ($\mathsf{has\_trap}$), it will record the trap type and execute this trap, then it will dispatch an instruction.

$$\frac{\mathsf{has\_trap}(Q) \qquad \mathsf{get\_tt}(Q) = w \qquad \mathsf{exe\_trap}(Q) = Q' \\ C_s \vdash (M_s, Q', D) \bullet\!\!\longrightarrow (M_s', Q'', D')}{(C_u, C_s) \vdash ((M_u, M_s), Q, D) \xRightarrow{w} ((M_u, M_s'), Q'', D')}$$

– For the following rule, if the machine does not have traps, it will select the code heap and the memory that belong to the current mode($\mathsf{usr\_mode}$ or $\mathsf{sup\_mode}$) and dispatch an instruction. Because the system is divided into user mode and supervisor mode, and the code heap and memory used by these modes are different, so before executing instruction scheduling, we need to select the corresponding code heap and memory according to the current mode.

$$\frac{\neg\mathsf{has\_trap}(Q) \qquad \mathsf{usr\_mode}(Q) \qquad C_u \vdash (M_u, Q, D) \bullet\!\!\longrightarrow (M_u', Q', D')}{(C_u, C_s) \vdash ((M_u, M_s), Q, D) \Longrightarrow ((M_u', M_s), Q', D')}$$

$$\frac{\neg\mathsf{has\_trap}(Q) \qquad \mathsf{sup\_mode}(Q) \qquad C_s \vdash (M_s, Q, D) \bullet\!\!\longrightarrow (M_s', Q', D')}{(C_u, C_s) \vdash ((M_u, M_s), Q, D) \Longrightarrow ((M_u, M_s'), Q', D')}$$

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. If the trap occurs, it will record the trap type into the event list. Otherwise e is $\perp$ for steps where no trap occurs. The transition of zero-or-multiple steps is defined as bellow.

$$\frac{}{\Delta \vdash S \models\!\!\xrightarrow{\textbf{nil}}{}^{0}\, S} \qquad \frac{\Delta \vdash S \xRightarrow{e} S'' \qquad \Delta \vdash S'' \models\!\!\xRightarrow{E}{}^{n}\, S'}{\Delta \vdash S \models\!\!\xRightarrow{e::E}{}^{n+1}\, S'}$$

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 determinative 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 $\Delta \vdash S \models\!\!\xRightarrow{E}{}^{*}\, S_1$ to represent zero-or-multiple steps of the execution under the given sequence of external interrupts E. Theorem 4.1 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 4.1 (Determinacy). *If* $\Delta \vdash S \models\!\!\xRightarrow{E}{}^{*}\, S_1, \Delta \vdash S \models\!\!\xRightarrow{E}{}^{*}\, S_2$*,then* $S_1 = S_2$. *where* $\Delta \vdash S \models\!\!\xRightarrow{E}{}^{*}\, S'$ *is defined as* $\exists n. \Delta \vdash S \models\!\!\xRightarrow{E}{}^{n}\, S'$.

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:

$$\Delta \vdash S \bullet\!\!\Longrightarrow^{n} S' \ \stackrel{def}{=\!=}\ \mathsf{usr\_mode}(S) \ \wedge\ \mathsf{empty\_DL}(S)\ \wedge\ \Delta \vdash S \models\!\!\xRightarrow{E}{}^{n}\, S'$$
$$\wedge\ \mathsf{no\_trap\_event}(E)$$

where

$$\mathsf{empty\_DL}(S) \ \stackrel{def}{=\!=}\ D = \textbf{nil} \qquad \textbf{where } S = ((M_u, M_s), Q, D)$$
$$\mathsf{no\_trap\_event}(E) \ \stackrel{def}{=\!=}\ \forall e \in E, e = \perp$$

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 (empty_DL), otherwise the system may enter the supervisor mode if there is a delayed write item in the delay list that will modify the S segment 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 4.2:

Theorem 4.2 (In User Mode). *If* $\Delta \vdash S \bullet\!\!\Longrightarrow^{n} S'$*, then* $\mathsf{usr\_mode}(S')$.

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.

After we have the Theorem 4.2, 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 as bellow:

Theorem 4.3 (Write Isolation). *If* $\Delta \vdash S \bullet\!\!\Longrightarrow^{n} S'$*, then* $\mathsf{sup\_part\_eq}(S, S')$.

where

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

Theorem 4.4 (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* $\mathsf{usr\_state\_eq}(S'_1, S'_2)$.

where

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

Theorem 4.3 says, if the system is running in the user mode, it does not modify the resource that belongs to the supervisor mode. Theorem 4.4 means 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 verify a trap handler, which is used to handle exception of the window overflow. The number of windows provided by SPARCv8 is finite. If we execute the **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 the trap. We give the code of the trap handler as below.

First, it takes the next window as the masked window, which is implemented by loop shift operation (Lines 1-5 and 7-10). Then the pointer (*cwp*) that always points to the current window points to the next window, and we store the value of the current window into the memory (Lines 6 and 11-26). Finally, the handler restores *cwp* and returns (Lines 27-30). The window overflow trap handler saves the oldest element of the window into the memory and makes the window available for the upcoming **save** operation.

```
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
```

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

$$\begin{aligned}
\mathsf{overflow\_pre\_cond}(W) \overset{def}{=\!=\!=} \; & \mathsf{single\_mask}(R(cwp), R(wim)) \;\wedge\; \mathsf{handler\_context}(R) \\
& \wedge\; \mathsf{normal\_cursor}(R) \;\wedge\; \mathsf{align\_context}(Q) \;\wedge \\
& \mathsf{set\_function}(R(pc), \mathsf{windowoverflow}, C_s) \;\wedge \\
& D = \mathbf{nil} \;\wedge\; \mathsf{length}(F) = 2N-3 \\
\mathbf{where}\; & W = (\varDelta, (\varPhi, Q, D)),\;\; \varDelta = (C_u, C_s),\;\; Q = (R, F) \\
\mathsf{overflow\_post\_cond}(W) \overset{def}{=\!=\!=} \; & \mathsf{single\_mask}(\mathsf{pre\_cwp}(2, R), R(wim)) \\
\mathbf{where}\; & W = (\varDelta, (\varPhi, Q, D)), Q = (R, F)
\end{aligned}$$

In the pre-condition, $\mathsf{single\_mask}(w, R(wim))$ indicates that the system simply masks the window w, and the rest of the window is all available. For example, the system must be in the supervisor mode, the trap must be disabled, and so on, as shown bellow.

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

$\mathsf{handler\_context}$ contains the unique state of the system after the $\mathsf{exe\_trap}$ function is executed.

$$\begin{aligned}
\mathsf{handler\_context}(R) \overset{def}{=\!=\!=} \; & 0 \le cwp \le 7 \;\wedge\; \mathsf{not\_annuled}(R) \;\wedge\; \mathsf{trap\_disabled}(R) \;\wedge \\
& \mathsf{no\_trap}(R) \;\wedge\; \mathsf{sup\_mode}(R)
\end{aligned}$$

$\mathsf{normal\_cursor}$ and $\mathsf{set\_function}$ illustrate the requirements for pc and npc before entering the overflow trap handler.

$$\mathsf{normal\_cursor}(R) \;\wedge\; \mathsf{set\_function}(R(pc), \mathsf{windowoverflow}, C_s)$$

where

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

$$\mathsf{normal\_cursor}(R) \quad \overset{def}{=\!=} \quad R(npc) = R(pc) + 4$$

$$\mathsf{set\_function}(a, j, C) \quad \overset{def}{=\!=} \quad \begin{cases} C(a) = \textbf{some } i \ \wedge \ \mathsf{set\_function}(a+4, j', C) & \textbf{if} \ \ j = i :: j' \\ \textbf{true} & \textbf{if} \ \ j = \textbf{nil} \end{cases}$$

align_context gives the requirements for word aligned of addresses.

$$\begin{aligned} \mathsf{align\_context}(Q) \quad \overset{def}{=\!=} \quad & \mathsf{word\_aligned}(R(l1)) \ \wedge \mathsf{word\_aligned}(R(l2)) \ \wedge \\ & \mathsf{word\_aligned}(R'(sp)) \\ \textbf{where} \ \ & Q = (R, F), \ (R', F') = \mathsf{right\_win}(1, (R, F)) \end{aligned}$$

The rest gives the requirements for the delay list and the frame list.

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 is no longer masked, which means the next window is available. In SPARCv8, when we execute the `save` instruction and enter the next window, the label of the window is decreased. So we use $\mathsf{pre\_cwp}(2, R)$ to represent the label of the window that has been masked, and it also means that we have an available window now.

Then 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 5.1, it says, if the initial state satisfies the precondition, then we can safely execute to 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 our Coq implementation [19].

Theorem 5.1 (Correctness of the Window Overflow Trap Handler).

If $\mathsf{overflow\_pre\_cond}(\Delta, S)$*, then exists S' and E, that* $\Delta \vdash S \overset{E}{\models\!\!\Longrightarrow}{}^{30} S'$*, and* $\mathsf{overflow\_post\_cond}(\Delta, S')$ *and* $\mathsf{no\_trap\_event}(E)$*.*

6 Coq Implementation

Section 3, Section 4 and Section 5 introduce the formal modeling of SPARCv8 ISA, the proof of our model, and the verification of the window overflow handler. These works are all implemented in Coq. The organization of the code is shown in Fig.5. The main idea of the style of our model comes from Compcert project. We also use its integer library and Map Library to facilitate the extensions of Compcert in the future. Because the hardware layer has a large number of integer computing work. We use the integer tactics library that comes from CertiC/OS-II verification project.

The following is the Coq code for modeling of assembly instructions. With this syntax, we can write specific assembly instructions in Coq. We also use the

File name	Description	Coq lines
Asm.v	Formalizing SPARCv8 ISA	947
Property.v	Properties and the proof	3033
WinOverflow.v	Verification of the window overflow trap handler	5761
Integers.v	Integer library (From CompCert)	4462
Maps.v	Map libaray (From CompCert)	204
Coqlib.v	Coq basic library (From CompCert)	1615
LibTactics.v	Some tactics in Coq (From Software Foundations)	4797
IntAuto.v	Some tactics for integer library (From CertiC/OS-II verification project)	673
MathSol.v	Some tactics for integer library (From CertiC/OS-II verification project)	775
Makefile	-	-
Total		22537

Fig. 5. The Verification Package

notation feature in Coq so that the syntax of our instructions in Coq is closer to the syntax of real SPARC assembly instructions.

```
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
 | ... .
end.
```

7 Conclusion and Future Work

In this paper, we have formalized the SPARCv8 instruction set in Coq, which provides the formal model for verifying SpaceOS at the assembly level. Also the formalization is able to help us to add SPARCv8 into the CompCert backend in the future. Since the correctness and availability are also critical in formal modeling of SPARCv8, we prove the determinacy and isolation properties to validate the model, and we also verify the window overflow handler to show the availability of our formalization.

For the future work, we will give the syntax and operational semantics of the remaining instructions, including integer arithmetic instructions, floating point instructions, and coprocessor instructions. To facilitate the code verification process, we will develop a program logic for reasoning about the assembly code, instead of doing verification in terms of the operational semantics directly. We hope to extend CompCert backend to support the SPARCv8 assembly language.

References

[1] Xu, F., Fu, M., Feng, X., Zhang, X., Zhang, H., Li, Z.: A Practical Verification Framework for Preemptive OS Kernels. In: International Conference on Computer Aided Verification, pp.59–79, Springer (2016)

[2] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T.:seL4: Formal verification of an OS kernel. In: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, pp.207–220. ACM (2009)

[3] SPARC, `https://en.wikipedia.org/wiki/SPARC`

[4] Qiao, L., Yang, M., Gu, B., Yang, H., Liu, B.: An Embedded Operating System Design for the Lunar Exploration Rover. In: Secure Software Integration & Reliability Improvement Companion (SSIRI-C), pp.160–165. IEEE (2011)

[5] Leroy, X.: Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In: ACM SIGPLAN Notices, vol. 41, no. 1, pp.42–54. ACM (2006)

[6] ARM, `https://en.wikipedia.org/wiki/ARM_architecture`

[7] x86, `https://en.wikipedia.org/wiki/X86`

[8] PowerPC, `https://en.wikipedia.org/wiki/PowerPC`

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

[10] The Coq Proof Assistant, `https://coq.inria.fr`

[11] Fox, A., Myreen, M.O.: A trustworthy monadic formalization of the ARMv7 instruction set architecture. In: International Conference on Interactive Theorem Proving, pp.243–258. Springer, Heidelberg (2010)

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

[13] Ssreflect, `http://ssr.msr-inria.inria.fr`

[14] Hou, Z., Sanan, D., Tiu, A., Liu, Y., Hoa, K.C.: An Executable Formalisation of the SPARCv8 Instruction Set Architecture: A Case Study for the LEON3 Processor. In: FM 2016: Formal Methods: 21st International Symposium, pp.388–405. Springer, Cyprus (2016)

[15] Feng, X., Shao, Z.: Modular verification of concurrent assembly code with dynamic thread creation and termination. In: ACM SIGPLAN Notices 40, no. 9, pp.254–267 (2005)

[16] Feng, X., Shao, Z., Vaynberg, A., Xiang, S., Ni, Z.: Modular verification of assembly code with stack-based control abstractions. In: ACM SIGPLAN Notices, vol. 41, no. 6, pp.401–414. ACM (2006)

[17] Feng, X., Shao, Z., Dong, Y., Guo, Y.: Certifying low-level programs with hardware interrupts and preemptive threads. In: ACM SIGPLAN Notices, vol. 43, no. 6, pp.170–182. ACM (2008)

[18] LEON3, `http://www.gaisler.com/index.php/products/processors/leon3`

[19] Formalizing SPARCv8 Instruction Set Architecture in Coq (Project Code), `https://github.com/wangjwchn/sparcv8-coq`