Formalizing SPARCv8 Instruction Set Architecture in Coq

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

[1] University of Science and Technology of China, Hefei, China
xyfeng@ustc.edu.cn

[2] Beijing Institute of Control Engineering, Beijing, China

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 systems running on SPARCv8 processors, one has to formalize the semantics of SPARCv8 ISA. In this paper, 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 a trap handler function 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 · Verification · Operational semantics

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 [20] and seL4 [17], 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 and stack. Therefore, the assembly code in OS kernels is not actually verified. To verify whether the assembly code satisfies its abstract specifications, it is inevitable to formalize the semantics of the assembly instructions.

This work is supported in part by grants from National Natural Science Foundation of China (NSFC) under Grant Nos. 61632005, 61379039 and 61502031.

As a highly efficient and reliable microprocessor, the SPARCv8 [6] instruction set architecture has been widely used in various processors for workstations, embedded systems, and space missions. For instance, SpaceOS [19] 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 [18] to compile SpaceOS. However, CompCert only supports translating Clight, which is an important subset of C, into ARM [1], x86 [10], PowerPC [5] 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 ISA. Our formal model is faithful to the behaviors of the instructions described in the SPARCv8 manual [7], including most of the features in SPARCv8, *e.g.*, windowed registers, delayed control transfer, interrupts 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 for 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 [2]. They contain around 11000 lines of coq scripts in total. The source code can be accessed via the link [3].

Related Work. Fox and Myreen gave the ARMv7 ISA model [14], 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 [16], and they used type classes, notations and the mathematics library Ssreflect [8]. 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 tool Isabelle [15], 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 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 can 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 [11–13], which give the formal models of different 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 [9]. It is widely used in the electronic systems of space devices for its high performance, high reliability and low power consumption. 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 `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` have register windows as their contexts, 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 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 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.

3.1 Syntax

Figure 1 shows the syntax of the SPARCv8 assembly language. Here we only give some typical instructions i that show the key features introduced in Sect. 2. **bicca** makes a delayed control transfer if the condition η holds, otherwise it annuls the next instruction and executes the following code (unless η is *al*, as explained in Sect. 3.3). The conditional expression η can be always (*al*), equal (*eq*), not equal (*ne*), *etc.*. **save** (or **restore**) saves (or restores) the caller's context by rotating the register window. **ticc** triggers a software trap, and **rett** returns from traps. **wr** writes some specific registers, which are defined as *Symbol*. *Symbol* 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* are used to store the processor's ancillary state. The write by **wr** may be delayed for several cycles, as explained in Sects. 3.2 and 3.3.

$$
\begin{array}{rlcl@{\qquad}rlcl}
(\mathit{Word}) & w & \in & \mathit{Int}32 & (\mathit{OpExp}) & \alpha & ::= & r \mid w \\
(\mathit{GenReg}) & r & ::= & r_0 \mid \ldots \mid r_{31} & (\mathit{AddrExp}) & \beta & ::= & \alpha \mid r + \alpha \\
(\mathit{AsReg}) & \mathit{asr} & ::= & \mathit{asr}_0 \mid \ldots \mid \mathit{asr}_{31} & (\mathit{TrapExp}) & \gamma & ::= & r \mid r + r \mid r + w \mid w \\
(\mathit{Symbol}) & \varsigma & ::= & \mathit{psr} \mid \mathit{wim} \mid \mathit{tbr} \mid y \mid \mathit{asr} & (\mathit{TestCond}) & \eta & ::= & \mathit{al} \mid \mathit{eq} \mid \mathit{ne} \mid \ldots \\
(\mathit{SparcIns}) & i & ::= & \multicolumn{5}{l}{\textbf{bicca}\ \eta\ \beta \mid \textbf{save}\ r_s\ \alpha\ r_d \mid \textbf{restore}\ r_s\ \alpha\ r_d \mid \textbf{rett}\ \beta \mid} \\
 & & & \multicolumn{5}{l}{\textbf{ticc}\ \eta\ \gamma \mid \textbf{wr}\ r_d\ \alpha\ \varsigma \mid \ldots}
\end{array}
$$

Fig. 1. The syntax of the SPARCv8 assembly language

The address expressions, operand expressions and trap expressions in these instructions are defined as *OpExp*, *AddrExp* and *TrapExp*. w stands for 32-bit integer constants (*Word*). r stands for general registers (*GenReg*).

Note that the `call`, `mov`, and `ret` instructions in the example in Sect. 2 are not given in the syntax, since they are all synthetic instructions, which can be defined from the basic instructions [7].

3.2 Machine States

Register File. Here we give the definition of the register files (*RegFile*).

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

We use q to represent the register name (*RegName*), including *GenReg* and *Symbol*, which were explained in Sect. 3.1. It also includes the program counter *pc*, the next program counter *npc*, the trap flag τ and annulling flag κ. A register file R is modeled as a total function mapping register names to 32-bit integers.

Program Counters. 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). The function next below defines the change of program counters when no transfer occurs. It updates *pc* with *npc* and increases *npc* by 4.

$$\mathsf{next}(R) \overset{\mathit{def}}{=\!=} R\{\mathit{pc} \rightsquigarrow R(\mathit{npc})\}\{\mathit{npc} \rightsquigarrow R(\mathit{npc}) + 4\}$$

If transfer occurs during the instruction execution, for example, if the evaluation of conditional expression returns **true** when we execute the instruction bicca, the function djmp will be executed:

$$\mathsf{djmp}(w, R) \overset{\mathit{def}}{=\!=} R\{\mathit{pc} \rightsquigarrow R(\mathit{npc})\}\{\mathit{npc} \rightsquigarrow w\}$$

djmp updates *pc* with *npc* and sets *npc* to the target address. As mentioned in the example in Sect. 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.

Window Registers. We use the frame and the frame list to describe the window registers and window rotating. The definitions are given as follows:

$$(\textit{Frame})\ \ f \ ::= \ [w_0, \ldots, w_7] \qquad (\textit{FrameList})\ \ F \ ::= \ \textbf{nil} \mid f{::}F$$
$$(\textit{RState})\ \ Q \ ::= \ (R, F)$$

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).

We divide the general registers ($r_0 \ldots r_{31}$) in the register file R into four groups, global out, local and in, as shown in Fig. 2(1). They represent the current view of the accessible general registers. There are also registers unaccessible, which are grouped into frames and stored on the frame list. We pair the register file and the frame list together as the register state Q.

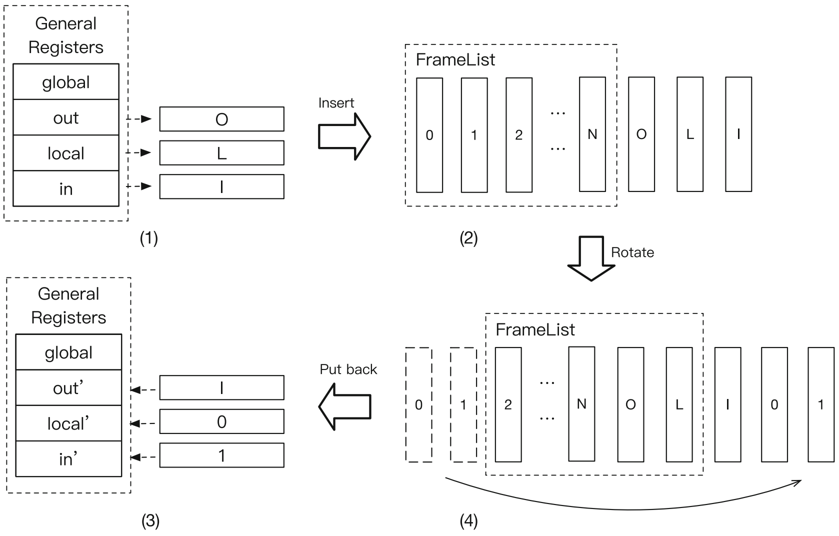

Fig. 2. Left rotation of the window

The view of currently accessible registers can be changed by rotating the window, which exchanges the data between the register file and the frame list. This is done to save and restore execution contexts, as what the `caller` and `sum3` do in the example in Sect. 2. Below we demonstrate the left rotation of the window in Fig. 2. The formal definition is given as left_win(Q) in Fig. 3. 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. 2(1). The conversion is formalized as fetch(R) in Fig. 3.
- As shown in Fig. 2(2) and (4), we can insert these 3 frames at the end of the frame list, then left rotate the frame list.

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

Fig. 3. Definition of the window rotation

– Finally, as shown in Fig. 2(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. 3.

Since the left rotation of the window increases the label of current window by 1, we also need to update the current window pointer (*cwp*, a segment of *psr*) to the new value, namely post_cwp.

Delayed Writes. When we execute the **wr** instruction to write the symbol register, the execution will be delayed for X cycles ($0 \leq X \leq 3$). The value of X is implementation-dependent. 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 value to be written.

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

There are 2 operations defined on the delay list, as shown bellow.

– When we execute the **wr** instruction, we will insert a delayed write into the delay list using function set_delay:

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

– At the beginning of each instruction cycle, we scan the delay list, remove the delayed writes whose delay cycles are 0 and execute them, and then decrement the delay cycles of the remaining delayed writes, as shown below:

$$\mathsf{exe\_delay}(Q,D) \overset{def}{=\!=} \begin{cases} (\mathsf{write\_symbol}(\varsigma,w,Q),D') & \textbf{if } D=(0,\varsigma,w)::D', \\ \textbf{let } (Q',D'') := \mathsf{exe\_delay}(Q,D') & \\ \ \textbf{in } (Q',(n-1,\varsigma,w)::D'') & \textbf{if } D=(n,\varsigma,w)::D', \\ & \quad n \neq 0 \\ (Q,D) & \textbf{otherwise} \end{cases}$$
$$\textbf{where } Q=(R,F)$$

$\mathsf{write\_symbol}$ writes the value w into the register ς. The details can be found in the technical report [4].

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.

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

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

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

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

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 some instructions such as **bicca**, **ticc**, *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 [7] or [15] 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 $(M,R) \xrightarrow{i} (M',R')$.

$$\begin{tikzcd}
\boxed{(M,R) \xrightarrow{i} (M',R')} \arrow[d, Rightarrow] & \text{Simple Instructions} \\
\boxed{(M,Q,D) \circ\!\xrightarrow{i} (M',Q',D')} \arrow[d, Rightarrow] & \text{Window Registers and Delayed Write} \\
\boxed{C \vdash (M,Q,D) \bullet\!\longrightarrow (M',Q',D')} \arrow[d, Rightarrow] & \text{Executing Delay and Handling Annulling Flag} \\
\boxed{\Delta \vdash S \xRightarrow{e} S'} & \text{Interrupts, Traps and Mode Switch}
\end{tikzcd}$$

Fig. 4. The structure of operational semantics

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 semantics of delay execution and annulling flag handling.

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 operational semantics of each layer. The omitted rules can be found in the technical report [4] and our Coq implementations [3].

Simple Instructions. The **bicca** 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 and whether to annul the next instruction by the conditional expression.

– If the value of the conditional expression is *false*, it makes no transfer and *sets the annulling flag* only.

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

– 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 intruction.

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

– If the type of the conditional expression is *al*, we will execute the delayed transfer and set the annulling flag. The rule is omitted here (see TR [4]).

Recall the definition of next and djmp in Sect. 3.2. Other functions not defined here can be found in the TR [4].

ticc $\eta\ \gamma$ evaluates the trap expression γ. If the condition η is true, it *sets the trap flag* ($\mathsf{set\_user\_trap}$). We use $w_{<6:0>}$ to represent the lowest 7 bits of w. **ticc** $\eta\ \gamma$ does nothing if η is false (the corresponding rule omitted).

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

Rules for other simple instructions are given in the TR [4].

Window Registers and Delayed Writes. Here we give semantics for instructions that manipulate the frame list and delay list. First, we use the LIFT1 rule to lift the transition $(M,R) \xrightarrow{i} (M',R')$ to $(M,Q,D) \circ\!\!\xrightarrow{i} (M',Q',D')$.

$$\frac{(M,R) \xrightarrow{i} (M',R')}{(M,(R,F),D) \circ\!\!\xrightarrow{i} (M',(R',F),D)} \ (\textsc{lift1})$$

When the **wr** $r_d\ \alpha\ \varsigma$ instruction is executed in user mode ($\mathsf{usr\_mode}$), since it does not have permissions for access the register wim, tbr and psr, so ς must be y or asr_i. Then it executes the XOR operation of α and r_d to get the value w. Next we insert the triple (X,ς,w) into the delay list D using the function $\mathsf{set\_delay}$ (see Sect. 3.2). Finally, it resets pc and npc with the function next.

$$\mathsf{inc\_win}(Q) \overset{def}{=\!=} \begin{cases} \mathsf{left\_win}(Q) & \textbf{if } \neg\mathsf{win\_masked}(\mathsf{post\_cwp}(R), R) \\ \bot & \textbf{otherwise} \end{cases}$$
$$\textbf{where } Q = (R,F), \mathsf{win\_masked}(w,R) \overset{def}{=\!=} 2^w \&\& R(\mathit{wim}) \neq 0$$

$$\mathsf{rett\_f}(Q) \overset{def}{=\!=} \begin{cases} (\mathsf{restore\_mode}(\mathsf{enable\_trap}(R')), F') & \textbf{if } \mathsf{inc\_win}(Q) \ = (R',F') \\ \bot & \textbf{otherwise} \end{cases}$$

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

$$\mathsf{tbr\_jmp}(R) \overset{def}{=\!=} R\{pc \rightsquigarrow R(\mathit{tbr})\}\{npc \rightsquigarrow R(\mathit{tbr}) + 4\}$$

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

Fig. 5. Auxiliary definitions

$$\frac{\textsf{usr\_mode}(R) \quad \varsigma = y \textbf{ or } asr_i \quad [\![\, r_d \,]\!]_R \textbf{ xor } [\![\, \alpha \,]\!]_R = w \quad D' = \textsf{set\_delay}(\varsigma, w, D)}{(M, (R, F), D) \overset{\textbf{wr } r_d \ \alpha \ \varsigma}{\circ\!\!\longrightarrow} (M, (\textsf{next}(R), F), D')} \ (\text{WRUSR})$$

When the **wr** r_d α ς instruction is executed in supervisor mode, it has fully access to all symbol registers. The rule of it is given in the TR [4].

For the rules SAVE and RESTORE, we first decrease or increase the label of the window using the function dec_win or inc_win. Then we evaluate the operand expression α to get the value a. Next we assign the value of $[\![\, r_s \,]\!]_R + a$ to r_d. The definition of the inc_win can be found in Fig. 5. The increasing operation is allowed if the post-window (post_cwp) is not masked ($\neg$win_masked). The function dec_win is similar to the inc_win and therefore it is not given here.

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

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

For the rule RETT, we first require that the trap is not enabled ($\neg$trap_enabled) and the system is in supervisor mode (sup_mode). Then we evaluate the address expression β to get the value w and require the address w to be *word-aligned*. Then we increase the label of the window (inc_win), enable the trap (enable_trap) and restore the previous mode (restore_mode) by using function rett_f, which is defined in Fig. 5. Finally, the system transfers to the address w by using djmp.

$$\frac{\begin{array}{ccc}\neg\textsf{trap\_enabled}(R) & \textsf{sup\_mode}(R) & [\![\, \beta \,]\!]_R = w \\ \textsf{word\_aligned}(w) & \textsf{rett\_f}(R, F) = (R', F') & \end{array}}{(M, (R, F), D) \overset{\textbf{rett } \beta}{\circ\!\!\longrightarrow} (M, (\textsf{djmp}(w, R'), F'), D)} \ (\text{RETT})$$

Exceptions. If some of the conditions (e.g., *word-aligned*) in the above rules 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 put the trap type label into the trap type register (a segment of *tbr*), then the system will execute this trap in the next cycle (see the explanation below). The abortions make the system to get stuck.

Executing Delay and Handling Annulling Flag. Here we check the delay list and handle the annulling flag.

– The exe_delay function executes the delayed writes (as described in Sect. 3.2). If the annulled flag has not been setted ($\neg$annulled), it will pick up an instruction from the code heap and execute it.

$$\frac{\begin{array}{cc}\textsf{exe\_delay}(Q, D) = (Q', D') & \neg\textsf{annulled}(Q') \\ C(Q'.pc) = i & (M, Q', D') \overset{i}{\circ\!\!\longrightarrow} (M', Q'', D'')\end{array}}{C \vdash (M, Q, D) \bullet\!\!\longrightarrow (M', Q'', D'')}$$

– Otherwise, If the annulled flag has been setted (annulled), it will skip one instruction and unset the annulling flag (clear_annul).

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

Interrupts, Traps and Mode Switch. In each instruction cycle, we deal with interrupts and traps first.

– If there is an interrupt request with level w and it is allowed (interrupt), the system triggers a trap after this external interrupt happens. It will record the trap type (get_tt) and execute this trap (exe_trap), then it will dispatch an instruction. The definition of interrupt and get_tt can be found in the technical report [4].

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

When we execute the trap (exe_trap), first we need to make sure that the system allows traps to occur (trap_enabled). Then we rotate the window to the right (right_win), forbid traps to occur (disable_trap), save the current mode (save_mode) and enter the supervisor mode (to_sup). Finally we save *pc* and *npc* to register r_{17} and r_{18} by using function save_pc_npc, unset the trap flag (clear_trap) and jump to the address of the trap handler (tbr_jmp). Function exe_trap, tbr_jmp and save_pc_npc are defined in Fig. 5. The function right_win is similar to the left_win and therefore it is not given here.

– If the system has a trap, it will record and execute this trap, and then dispatch an instruction.

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

– If the system does not have traps, it will select the code heap and the memory according to the mode (usr_mode or sup_mode) and dispatch an instruction.

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

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

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 $\bot$ for steps where no trap occurs. The transition of zero-or-multiple steps is defined as below.

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

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 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 $\varDelta \vdash S \models\!\!\xRightarrow{E}{}^{*}\, S_1$ 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* $\varDelta \vdash S \models\!\!\xRightarrow{E}{}^{*}\, S_1$, $\varDelta \vdash S \models\!\!\xRightarrow{E}{}^{*}\, S_2$, *then* $S_1 = S_2$. *where* $\varDelta \vdash S \models\!\!\xRightarrow{E}{}^{*}\, S'$ *is defined as* $\exists n, \varDelta \vdash S \models\!\!\xRightarrow{E}{}^{n}\, S'$.

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:

$$\begin{aligned}\varDelta \vdash S \bullet\!\!\Longrightarrow^{n} S' \;\overset{def}{=\!=}\; & \mathsf{usr\_mode}(S) \;\wedge\; \mathsf{empty\_DL}(S) \;\wedge\; \varDelta \vdash S \models\!\!\xRightarrow{E}{}^{n}\, S' \\ & \wedge\; \mathsf{no\_trap\_event}(E)\end{aligned}$$

where

$$\begin{aligned}\mathsf{empty\_DL}(S) \quad & \overset{def}{=\!=} \quad D = \textbf{nil} \qquad \textbf{where } S = ((M_u, M_s), Q, D) \\ \mathsf{no\_trap\_event}(E) \quad & \overset{def}{=\!=} \quad \forall e \in E, e = \bot\end{aligned}$$

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.

Based on 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 bellow:

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

where

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

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

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

Theorem 4.3 shows that if the system is running in the user mode, it does not modify the resource that belongs to the supervisor mode. Theorem 4.4 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 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, the handler takes the next window as the masked window, which is implemented by loop shift operation (Lines 1–5 and 7–10). Then the pointer (named *cwp*, a segment of *psr*) 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 first 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. $\mathsf{pre\_cwp}(n, R)$ gives the window in front of the current window and the distance of them is n. 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.

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, which 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 the technical report [4] and our Coq implementation [3].

Theorem 5.1 (Correctness of the Window Overflow Trap Handler). *If* $\mathsf{overflow\_pre\_cond}(\Delta, S)$*, then forall S' and E, if* $\Delta \vdash S \overset{E}{\Longmapsto}{}^{30} S'$*, then* $\mathsf{overflow\_post\_cond}(\Delta, S')$ *and* $\mathsf{no\_trap\_event}(E)$*.*

6 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 can help us to add SPARCv8 into the backend of CompCert 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. Arm architecture. https://en.wikipedia.org/wiki/ARM_architecture
2. The coq proof assistant. https://coq.inria.fr
3. Formalizing sparcv8 instruction set architecture in coq (project code). https://github.com/wangjwchn/sparcv8-coq
4. Formalizing sparcv8 instruction set architecture in coq (technical report). https://wangjwchn.github.io/pdf/sparc-coq-tr.pdf
5. Powerpc. https://en.wikipedia.org/wiki/PowerPC
6. Sparc. https://en.wikipedia.org/wiki/SPARC
7. The sparc architecture manual v8. http://gaisler.com/doc/sparcv8.pdf
8. Ssreflect. http://ssr.msr-inria.inria.fr
9. Sun microsystems. https://en.wikipedia.org/wiki/Sun_Microsystems
10. x86. https://en.wikipedia.org/wiki/X86
11. Feng, X., Shao, Z.: Modular verification of concurrent assembly code with dynamic thread creation and termination. In: International Conference on Functional Programming (ICFP), pp. 254–267. ACM (2005)

12. Feng, X., Shao, Z., Dong, Y., Guo, Y.: Certifying low-level programs with hardware interrupts and preemptive threads. In: Conference on Programming Language Design and Implementation (PLDI), pp. 170–182. ACM (2008)
13. Feng, X., Shao, Z., Vaynberg, A., Xiang, S., Ni, Z.: Modular verification of assembly code with stack-based control abstractions. In: Conference on Programming Language Design and Implementation (PLDI), pp. 401–414. ACM (2006)
14. Fox, A., Myreen, M.O.: A trustworthy monadic formalization of the ARMv7 instruction set architecture. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 243–258. Springer, Heidelberg (2010). doi:10.1007/978-3-642-14052-5_18
15. 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: Fitzgerald, J., Heitmeyer, C., Gnesi, S., Philippou, A. (eds.) FM 2016. LNCS, vol. 9995, pp. 388–405. Springer, Cham (2016). doi:10.1007/978-3-319-48989-6_24
16. 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 (PPDP), pp. 13–24. ACM (2013)
17. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., et al.: sel4: formal verification of an os kernel. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP), pp. 207–220. ACM (2009)
18. Leroy, X.: Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In: 33rd Symposium Principles of Programming Languages (POPL), pp. 42–54. ACM (2006)
19. Qiao, L., Yang, M., Gu, B., Yang, H., Liu, B.: 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), pp. 160–165. IEEE Computer Society (2011)
20. Xu, F., Fu, M., Feng, X., Zhang, X., Zhang, H., Li, Z.: A practical verification framework for preemptive OS kernels. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 59–79. Springer, Cham (2016). doi:10.1007/978-3-319-41540-6_4