Journal of computer science and technology: Instruction for authors. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY

Modular Verification of SPARCv8 Code

Abstract Inline assembly code is common in system software to interact with the underlying hardware platforms. Safety and correctness of the assembly code is crucial to guarantee the safety of the whole system. In this paper, we propose a practical Hoare-style program logic for verifying SPARC assembly code. The logic supports modular reasoning about the main features of SPARCv8 ISA, including delayed control transfers, delayed writes to special registers, and register windows. It also supports relational reasoning for refinement verification and we have applied it to verify that there is a contextual refinement between a context switch routine in SPARCv8 and switch primitive. The program logic and its soundness proof have been mechanized in Coq.

Keywords SPARCv8, assembly code verification, context switch, Coq, refinement verification

1 Introduction

Operating system kernels are at the most foundational layer of computer software systems. To interact directly with hardware, many important components in OS kernels are implemented in assembly, such as the context switch code or the code that manages interrupts. In addition, there is also code written directly in assembly to achieve high performance (*e.g.* memcpy in linux v2.6.17.10 [1]). Correctness of such assembly code is crucial to ensure the safety and security of the whole system. However, assembly code verification remains a challenging task in existing work on OS kernel verification (*e.g.* [2, 3, 4]), where the assembly code is either unverified or verified based on operational semantics without a general program logic.

SPARC (Scalable Processor ARChitecture) is a CPU instruction set architecture (ISA) with high-performance and great flexibility [5]. It has been widely used in various processors for workstations and embedded systems. The SPARCv8 ISA has some interesting features, which make it a non-trivial task to design a Hoare-style program logic for assembly code.

- *Delayed control transfers.* SPARCv8 has two program counters `pc` and `npc`. The `npc` register points to the next instruction to run. Control-transfer instructions in SPARCv8 change `npc` instead of `pc` to the target program point, while `pc` takes the original value of `npc`. This makes the control transfer to happen one cycle later than the execution of the control transfer instructions.
- *Delayed writes.* The `wr` instruction that writes a special class of registers such as the window invalid mask register `wim` does not take effect immediately. That is, "they may take until completion of the third instruction following the write instruction to consummate their write operation. The number of delay instructions (0 to 3) is implementation-dependent" [5].
- *Register windows.* SPARCv8 uses register windows and a window rotation mechanism to avoid saving contexts in the stack directly and achieves high performance in context management.

We use a simple example in Fig. 1 to show these three features. SPARCv8 has 32 general registers,

Regular Paper

which are split into four logic groups as global ($\mathtt{r}_0 \sim \mathtt{r}_7$), out ($\mathtt{r}_8 \sim \mathtt{r}_{15}$), local ($\mathtt{r}_{16} \sim \mathtt{r}_{23}$) and in ($\mathtt{r}_{24} \sim \mathtt{r}_{31}$) registers. Correspondingly, we give aliases "$\%\mathtt{g}_0 \sim \%\mathtt{g}_7$", "$\%\mathtt{o}_0 \sim \%\mathtt{o}_7$", "$\%\mathtt{l}_0 \sim \%\mathtt{l}_7$" and "$\%\mathtt{i}_0 \sim \%\mathtt{i}_7$" for these groups respectively.

```
CALLER :                          ChangeY :
    ...                       5   rd   Y, %l0
1   mov  1, %o0               6   wr   %i0, 0, Y
2   call  ChangeY             7   nop
3   save  %sp, −64, %sp       8   nop
4   mov  %o0, %l0             9   nop
    ...                       10  ret
                              11  restore   %l0, 0, %o0
```

Fig.1. An Example for SPARC Code

CALLER in Fig. 1 calls ChangeY, which updates the *special register* Y and returns its original value.

ChangeY requires an input parameter as the new value for the special register Y. CALLER calls ChangeY at line 2, and pc and npc point to line 2 and 3 respectively at this moment. The call instruction changes the value of pc to npc and lets npc point to the entry of ChangeY at line 5, which means the control-flow will not transfer to ChangeY in the next cycle, but in the cycle after the execution of the save instruction following the call. Similarly, when ChangeY returns (at line 10), the control is transferred back to the caller after executing the restore instruction at line 11. We call this feature "delayed control transfers".

SPARCv8 uses the save instruction (at line 3 in the example) to save the current context and restore (at line 10) to restore it. As explained above, among the 32 general registers, the out, local and in registers form the *current register window*. The local registers are for private use in the current context. The in and out registers are shared with adjacent register windows for parameters passing. The save instruction rotates the register window from the current one to the next. Then the local and in registers in the original window are no longer accessible, and the original out registers become the in registers in the current window. In Fig. 1, the CALLER uses the save instruction (at line 3) to save its context (local and in registers) and rotate the register window (so that the out registers become the in registers). So, the $\%\mathtt{i}_0$ register assessed in ChangeY at line 6 is the same register as the $\%\mathtt{o}_0$ modified in CALLER at line 1. The restore instruction does the inverse. The arguments taken by the save and restore instructions are irrelevant here and can be ignored.

At line 6, the wr instruction tries to update the special register Y with the value of $\%\mathtt{i}_0 \oplus 0$ (bitwise exclusive OR). Note that the $\%\mathtt{i}_0$ register here is the same register as $\%\mathtt{o}_0$ at line 1. However, the write is delayed for X cycles, where X is some predefined system parameter that ranges from 0 to 3. For portability, programmers usually do not rely on the exact value of X and assume it takes the maximum value 3. Therefore three nop instructions are inserted. Reading of Y earlier than line 9 may give us the old value. This feature is called "delayed writes".

These features make the semantics of the SPARCv8 code context-dependent. For instance, a read of a special register (*e.g.* the register Y in the above example) needs to make sure there are enough instructions executed since the most recent *delayed* write. As another example, the instruction following the call can be any instruction in general, but it is not supposed to update the register $\mathtt{r}_{15}$, which contains the return address saved by the call instruction. In addition, the delayed control transfer and the register windows also allow highly flexible calling conventions. Together, they make it a challenging task to have a Hoare-style program logic for local and modular reasoning of SPARCv8 assembly code.

Working towards a certified OS kernel for aerospace crafts whose inline assembly is written in SPARCv8, we

try to address these challenges and propose a practical program logic for realistically modelled SPARCv8 code. We have applied our logic to verify the main body of the task context switch routine in the kernel [6].

However, the OS kernel is implemented as C language mainly and SPARCv8 as inline assembly. Just having a traditional Hoare-style program logic for SPARCv8, which can only make sure the safe execution of SPARCv8 program if the initial state satisfies the precondition, is insufficient. Xu *et al.* [2] propose a program logic for verifying the correctness of OS kernel implemented in C language with inline assembly, but they use *abstract assembly primitives* to substitute the inline assembly in their verification work. As a supplement to their work, we extend our program logic so that it can ensure the contextual refinement relation, shown in (1), between the implementations and their corresponding abstract assembly primitives. Here, we use $\mathbb{C}$, A, and C_{as} to represent the C language program, the set of abstract assembly primitives and the implementations of abstract assembly primitives respectively. It means C_{as} refines A under *any context* $\mathbb{C}$.

$$\forall\, \mathbb{C}.\, \mathbb{C}[C_{\text{as}}] \subseteq \mathbb{C}[\text{A}] \tag{1}$$

Here we use "$\subseteq$" to represent the refinement relation.

However, if we use C program as a client code to call inline assembly code, we need to define the semantics of C-assembly interaction.

$$\begin{array}{l} [\![\ \mathbb{C}[\text{A}]\ ]\!]^{\mathsf{C}} \\ \textcircled{1}\ \cup\mathsf{I} \iff \boxed{C = \mathit{Comp}(\mathbb{C})} \\ [\![\ C[\Omega]\ ]\!]^{\mathsf{P\text{-}SPARCv8}} \\ \textcircled{2}\ \cup\mathsf{I} \iff \boxed{\vdash C_{\text{as}} : \Omega} \\ [\![\ C[C_{\text{as}}]\ ]\!]^{\mathsf{SPARCv8}} \end{array}$$

Fig.2. Idea to establish contextual refinement

Since the goal of this paper is to verify the correctness of the assembly code with respect to the abstract assembly primitives, we want to avoid the C-assembly interaction (and leave it as future work). We decompose the OS verification tasks into two steps, as shown in Fig. 2, and focus on step ② only in this paper.

The source program of OS kernel $\mathbb{C}$[A], which executes under the C language semantics (shown as $[\![\ \_\ ]\!]^{\mathsf{C}}$), is implemented as C language with a set of abstract assembly primitive A. The compiler (*Comp*) translates the C program $\mathbb{C}$ to SPARCv8 code C. As ① shows, we assume the compilation ensures the refinement between $\mathbb{C}$[A] and $C[\Omega]$ that executes under Pseudo-SPARCv8 semantics shown as $[\![\ \_\ ]\!]^{\mathsf{P\text{-}SPARCv8}}$. Here, the Ω represents the set of abstract assembly primitives in the middle layer. We use distinguished notations to represent the set of abstract assembly primitives in source and intermediate level, because they execute on different program states and have different semantics. The Pseudo-SPARCv8 language $C[\Omega]$, which uses SPARCv8 as client code and is able to call abstract assembly primitives in Ω, will be defined in the following sections. In step ②, we verify using our refinement logic that the whole SPARCv8 program $C[C_{\text{as}}]$ executing under the realistically modelled SPARCv8 semantics, represented as $[\![\ \_\ ]\!]^{\mathsf{SPARCv8}}$, refines the program $C[\Omega]$ executing under the Pseudo-SPARCv8 semantics. Finally, we can get $[\![\ C[C_{\text{as}}]\ ]\!]^{\mathsf{SPARCv8}} \subseteq [\![\ \mathbb{C}[\text{A}]\ ]\!]^{\mathsf{C}}$. In this work, we focus on step ②, and leave step ① as future work.

Our work is based on earlier work on assembly code verification but makes the following contributions:

- We propose a new program logic which supports relational reasoning for refinement verification. It ensures that a verified SPARCv8 function contextually refines its corresponding abstract assembly primitive. Our logic supports all the above features of SPARCv8. We redefine basic blocks to include the instruction following the jump or return

as the tail of a block, which models the delayed control transfer. To reason about delayed writes, we introduce a modal assertion $\triangleright_t \mathtt{sr} \mapsto w$, saying that the special register `sr` will hold the value w in up to t cycles. We also give logic rules for `save` and `restore` instructions that do register window rotation.

- In order to support refinement verification, we define a Pseudo-SPARCv8 language as the language to implement the high-level specification. It also hides the details of sophisticated register window mechanism in SPARCv8 by abstraction, and makes its language model simpler than realistic SPARCv8. So, it can provide some convenience to write the abstract assembly primitive and reason in Pseudo-SPARCv8 level.

- Following SCAP [7], our logic supports modular reasoning of function calls in a direct-style. However, we use the traditional pre- and post-conditions as function specifications, instead of the assertion g used in SCAP. This allows us to reuse existing techniques (*e.g.* Coq tactics) to simplify the verification process. The logic rules for function call and return are general and independent of any specific calling convention.

- We give direct-style semantic interpretation for the logic judgments, based on which we establish the soundness. This is different from previous work, which either does syntactic-based soundness proof (*e.g.* SCAP [7]) or treats return code pointers as first-class code pointers and gives CPS-style (continuation-passing style) semantics. Those approaches for soundness make it difficult to verify the interaction between the inline assembly and the C code in the kernel, the latter being verified following a direct-style program logic.

- Context switch of concurrent tasks is an important component in OS kernels. It is usually implemented as inline assembly because of the need to access registers and the stack. We apply our logic and verify that there is a contextual refinement between a context switch routine implemented in SPARCv8 and an abstract switch primitive.

The program logic and its soundness proof have been mechanized in Coq [8].

This paper extends the conference paper in APLAS 2018 [6]. The program logic there can only verify the partial correctness of SPARCv8 code and we make the following expansions:

- We propose a new program logic to do relational reasoning for refinement verification (Sec. 4.4 for details).

- In order to support refinement verification, this paper presents a new Pseudo-SPARCv8 language as the high-level specification language (Sec. 4.1 for details).

- We also use the new logic to verify the implementation of a context switch routine, by showing that the implementation contextually refines an abstract switch primitive (Sec. 5 for details).

In the remainder of the paper, we present the program model and operational semantics of SPARCv8 in Sec. 2. For clear presentation, we use a simplified version of our program logic that doesn't support refinement verification to demonstrate how our logic supports the three main features of SPARCv8 in Sec. 3, which is the main point of our work and irrelevant with refinement verification. We present the Pseudo-SPARCv8 program and the relational program logic for refinement reasoning in Sec. 4. We show the verification of a context switch routine in SPARCv8 in Sec. 5. Finally, we discuss more on related work and conclude in Sec. 6.

$$\begin{array}{llll}
\text{(Word)}\ w,\mathtt{f} \in \text{Int32} & \text{(Block)}\ b \in \mathbb{Z} & \text{(Addr)}\ l \in \text{Block} \times \text{Word} & \text{(Val)}\ v ::= w \mid l \\
\\
\text{(Prog)}\ \mathrm{P} ::= (C, S, \mathtt{pc}, \mathtt{npc}) & & \text{(CodeHeap)}\ C \in \text{Word} \rightharpoonup \text{Comm} & \\
\text{(State)}\ S ::= (M, Q, D) & & \text{(RState)}\ Q ::= (R, F) & \\
\text{(Mem)}\ M \in \text{Addr} \rightharpoonup \text{Val} & & \text{(ProgCount)}\ \mathtt{pc}, \mathtt{npc} \in \text{Word} & \\
\text{(OpExp)}\ \mathtt{o} ::= \mathtt{r} \mid w & & \text{(AddrExp)}\ \mathtt{a} ::= \mathtt{o} \mid \mathtt{r} + \mathtt{o} &
\end{array}$$

$$\begin{array}{ll}
\text{(Comm)}\ c ::= & \mathtt{i} \mid \mathtt{call\ f} \mid \mathtt{jmp\ a} \mid \mathtt{retl} \mid \mathtt{be\ f} \\
\text{(SimpIns)}\ \mathtt{i} ::= & \mathtt{ld\ a\ r}_d \mid \mathtt{st\ r}_s\ \mathtt{a} \mid \mathtt{nop} \mid \mathtt{add\ r}_s\ \mathtt{o\ r}_d \mid \mathtt{save\ r}_s\ \mathtt{o\ r}_d \mid \mathtt{restore\ r}_s\ \mathtt{o\ r}_d \\
& \mid \mathtt{rd\ sr\ r}_d \mid \mathtt{wr\ r}_s\ \mathtt{o\ sr} \mid \ldots \\
\text{(InstrSeq)}\ \mathbb{I} ::= & \mathtt{i};\mathbb{I} \mid \mathtt{jmp\ a};\ \mathtt{i} \mid \mathtt{call\ f};\mathtt{i};\mathbb{I} \mid \mathtt{retl\ i} \mid \mathtt{be\ f};\ \mathtt{i};\ \mathbb{I}
\end{array}$$

Fig.3. Machine States and Language for SPARCv8 Code

2 The SPARCv8 Assembly Language

We introduce the key SPARCv8 instructions, the model of machine states, and the operational semantics in this section.

2.1 Language syntax and states

The machine model and syntax of SPARCv8 assembly language are defined in Fig. 3. Here, we follow the block-based memory [9] introduced in CompCert to define the memory in our work. The memory address l is defined as a pair of its block id and the offset. The type of block (Block) is the integer in mathemantics presented as $\mathbb{Z}$, and the type of offset (Word) is a 32-bit integer, which we called *words* in our work. So, the value (Val) here is either a word w or address l. The whole program configuration (Prog) P consists of the code heap (CodeHeap) C, the machine state (State) S, and the program counters `pc` and `npc`. The code heap C is a partial function from labels `f` to commands c. Labels are also 32-bit integers (called *words*), which can be viewed as addresses or locations where the commands are saved in the code heap. The operand expression (OpExp) `o`, which is either a general register `r` or a word w, and address expression (AddrExp) `a`, which is either an operand expression or a sum of the values of register `r` and an operation expression, are auxiliary definitions used as parameters of commands. Commands (Comm) in SPARCv8 can be classified into two categories: (1) simple instruction (SimpIns) `i`, which does sequential operation, *e.g.* arithmetic operation "`add`", or memory operations "`ld`" (load) and "`st`" (store), or register window operations "`save`" and "`restore`", or special register operations "`rd`" (read) and "`wr`" (write), or "`nop`", whose execution changes no program state (except assigning `npc` to `pc` and (`npc`+4) to `npc`); (2) control-transfer instructions, *e.g.* `call` and `retl` for function call and return, or `jmp` and `be` for unconditional and conditional branch.

The machine state (State) S consists of three parts: the memory (Mem) M, the register state (Rstate) Q which is a pair of register file (RegFile) R and frame list (FrameList) F, and the delay buffer (DelayBuff) D. As defined in Fig. 4, R is a partial mapping from register names (RegName) to values. Registers include the general registers `r`, the processor state register (PsrReg) `psr` and the special registers (SpeReg) `sr`. The processor state register `psr` contains the integer condition code fields `n`, `z`, `v` and `c`, which can be modified by the arithmetic and logical instructions and used for conditional control-transfer, and `cwp` recording the id of the current register window. We explain the frame list F and the delay buffer D below.

$$\begin{array}{rrcl@{\qquad}rrcl}
\text{(RegFile)} & R & \in & \text{RegName} \rightharpoonup \text{Val} & \text{(RegName)} & \texttt{rn} & ::= & \texttt{r}_0 \mid \ldots \mid \texttt{r}_{31} \mid \texttt{psr} \mid \texttt{sr} \\
\text{(PsrReg)} & \texttt{psr} & ::= & \texttt{n} \mid \texttt{z} \mid \texttt{v} \mid \texttt{c} \mid \texttt{cwp} & \text{(SpeReg)} & \texttt{sr} & ::= & \texttt{wim} \mid \texttt{Y} \mid \texttt{asr}_0 \mid \ldots \mid \texttt{asr}_{31} \\
\text{(FrameList)} & F & ::= & \text{nil} \mid \text{fm} :: F & \text{(Frame)} & \text{fm} & ::= & [v_0, \ldots, v_7] \\
\text{(DelayBuff)} & D & ::= & \text{nil} \mid (t, \texttt{sr}, w) :: D & \text{(DelayCycle)} & t & \in & \{0, 1, \ldots, X\}
\end{array}$$

Fig.4. Register File, Frame List and DelayBuffer

Register windows and frame List. SPARCv8 provides 32 general registers that are split into four groups as global ($\texttt{r}_0 \sim \texttt{r}_7$), out ($\texttt{r}_8 \sim \texttt{r}_{15}$), local($\texttt{r}_{16} \sim \texttt{r}_{23}$) and in ($\texttt{r}_{24} \sim \texttt{r}_{31}$) registers. The latter three groups (out, local and in) form the current *register window*.

At the entry and exit of functions and traps, one may need to save and restore some of the general registers as execution contexts. Instead of saving them into stacks in memory, SPARCv8 uses multiple register windows to form a circular stack, and does window rotation for efficient context save and restore. As shown in Fig. 5, there are N register windows ($N = 8$ here) consisting of $2 \times N$ groups of registers (each group containing 8 registers). The cwp register (part of psr) records the id number of the current window ($\texttt{cwp} = 0$ in this example).

The in and out registers of each window are shared with its adjacent windows for parameter passing. For example, the in registers of the w_0 is the out registers of the w_1, and the out registers of the w_0 is the in registers of the w_7. This explains why we need only $2 \times N$ groups of registers for N windows, while each window consisting of three groups (out, local and in).

To save the context, the save instruction rotates the window by decrementing the cwp pointer (modulo N). So w_7 becomes the current window. The out registers of w_0 becomes the in registers of w_7. The in and local registers of w_0 become inaccessible. This is like pushing them onto the circular stack. The restore instruction does the inverse, which is like a stack pop.

The wim register is used as a bit vector to record the end of the stack. Each bit in wim corresponds to a register window. The bit corresponding to the last available window is set to 1, which means *invalid*. All other bits are 0 (*i.e. valid*). When executing save (and restore), we need to ensure the next window is valid, in order to avoid the overflow of register window because of the limitation of the number of windows. We use the assertion $\textbf{win\_valid}(w_{id}, R)$ defined in Fig. 6 to say the window pointed to by w_{id} is valid, given the value of wim in R.

We use the frame list F to model the circular stack consisting of register windows. As defined in Fig. 4, a frame is an array of 8 words, modeling a group of 8 registers. F consists of a sequence of frames corresponding to all the register windows except the out, local and in registers in the current window. Then save saves the local and in registers onto the head of F and loads the two groups of register at the *tail* of F to the local and out registers (and the original out registers becomes the in group). The restore instruction does the inverse. The operations are defined formally in Fig. 6. Here, we use " :: " for adding an element at the head of a list, and use "." for appending an element at the tail of a list.

The delay buffer. The delay buffer D is a sequence of delayed writes. Because the wr instruction does not update the target register immediately, we put the write operation onto the delay buffer. A delayed write is recorded as a triple consisting of the remaining cycles t to be delayed, the target special register sr and the value w to be written. Note that we restrict that the value of a special register can only be a word, because the special registers are used to record the state of pro-

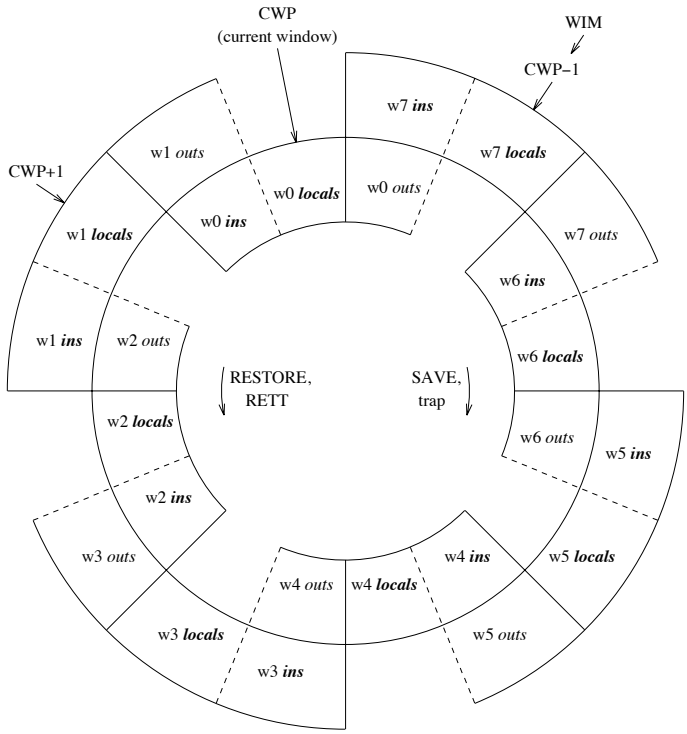


Fig.5. Register Windows (figure taken from [5])

$$\mathsf{out} ::= [\mathtt{r}_8, \ldots, \mathtt{r}_{15}] \qquad \mathsf{local} ::= [\mathtt{r}_{16}, \ldots, \mathtt{r}_{23}] \qquad \mathsf{in} ::= [\mathtt{r}_{24}, \ldots, \mathtt{r}_{31}]$$

$$R([\mathtt{r}_i, \ldots, \mathtt{r}_{i+k}]) ::= [R(\mathtt{r}_i), \ldots, R(\mathtt{r}_{i+k})]$$

$$R\{[\mathtt{r}_i, \ldots, \mathtt{r}_{i+7}] \rightsquigarrow \mathrm{fm}\} ::= R\{\mathtt{r}_i \rightsquigarrow v_0\} \ldots \{\mathtt{r}_{i+7} \rightsquigarrow v_7\} \qquad \text{where} \quad \mathrm{fm} = [v_0, \ldots, v_7]$$

$$\mathbf{win\_valid}(w_{id}, R) ::= 2^{w_{id}} \,\&\, R(\mathtt{wim}) = 0$$
where & is the bitwise AND operation.

$$\mathbf{next\_cwp}(w_{id}) ::= (w_{id} + N - 1)\%N \qquad \mathbf{prev\_cwp}(w_{id}) ::= (w_{id} + 1)\%N$$

$$\mathbf{save}(R, F) ::= \begin{cases} (R', F') & \text{if } w'_{id} = \mathbf{next\_cwp}(R(\mathtt{cwp})), \mathbf{win\_valid}(w'_{id}, R), \\ & \quad F = F'' \cdot \mathrm{fm}_1 \cdot \mathrm{fm}_2,\ F' = R(\mathsf{local}) :: R(\mathsf{in}) :: F'', \\ & \quad R'' = R\{\mathsf{in} \rightsquigarrow R(\mathsf{out}), \mathsf{local} \rightsquigarrow \mathrm{fm}_2, \mathsf{out} \rightsquigarrow \mathrm{fm}_1\}, \\ & \quad R' = R''\{\mathtt{cwp} \rightsquigarrow w'_{id}\}, \\ \bot & \text{if } \neg\mathbf{win\_valid}(\mathbf{next\_cwp}(R(\mathtt{cwp})), R) \end{cases}$$

$$\mathbf{restore}(R, F) ::= \begin{cases} (R', F') & \text{if } w'_{id} = \mathbf{prev\_cwp}(R(\mathtt{cwp})), \mathbf{win\_valid}(w'_{id}, R), \\ & \quad F = \mathrm{fm}_1 :: \mathrm{fm}_2 :: F'',\ F' = F'' \cdot R(\mathsf{out}) \cdot R(\mathsf{local}), \\ & \quad R'' = R\{\mathsf{in} \rightsquigarrow \mathrm{fm}_2, \mathsf{local} \rightsquigarrow \mathrm{fm}_1, \mathsf{out} \rightsquigarrow R(\mathsf{in})\}, \\ & \quad R' = R''\{\mathtt{cwp} \rightsquigarrow w'_{id}\}, \\ \bot & \text{if } \neg\mathbf{win\_valid}(\mathbf{prev\_cwp}(R(\mathtt{cwp})), R) \end{cases}$$

Fig.6. Auxiliary Definitions for Instruction `save` and `restore`

cessor, and it is impossible to store memory addresses in them.

Instruction sequences. We use an instruction sequence $\mathbb{I}$ to model a basic block, *i.e.* a sequence of commands ending with a control transfer. As defined in Fig. 3, we require that a delayed control-transfer in-

struction must be followed by a simple instruction `i`, because the actual control-transfer occurs after the execution of `i`. The end of each instruction sequence can only be `jmp` or `retl` followed by a simple instruction `i`. Note that we do not view the `call` instruction as the end of a basic block, since the callee is expected to return, following our direct-style semantics for function calls. We define $C[\mathtt{f}]$ to extract an instruction sequence starting from `f` in C below.

$$C[\mathtt{f}] = \begin{cases} \mathtt{i};\mathbb{I} & C(\mathtt{f}) = \mathtt{i} \text{ and } C[\mathtt{f}+4] = \mathbb{I} \\ c;\mathtt{i} & c = C(\mathtt{f}) \text{ and } c = \mathtt{jmp\ a} \text{ or } \mathtt{retl} \\ & \text{and } C(\mathtt{f}+4) = \mathtt{i} \\ c;\mathtt{i};\mathbb{I} & c = C(\mathtt{f}) \text{ and } c = \mathtt{call\ f} \text{ or } \mathtt{be\ f} \\ & \text{and } C(\mathtt{f}+4) = \mathtt{i} \text{ and } C[\mathtt{f}+8] = \mathbb{I} \\ \textbf{undefined} & \text{otherwise} \end{cases}$$

2.2 Operational Semantics

The operational semantics is taken from Wang *et al.* [10], but we use a block-based memory model and omit features like interrupts and traps. We show the selected rules in Fig. 7. The program transition relation $(C, S, \mathtt{pc}, \mathtt{npc}) ::\Longrightarrow (C, S', \mathtt{pc}', \mathtt{npc}')$ is defined in Fig. 7 (a). Before the execution of the instruction pointed by `pc`, the delayed writes in D with 0 delay cycles are executed first. The execution of the delayed writes are defined in the form of $(R, D) \rightrightarrows (R', D')$ below:

$$\frac{}{(R, \mathrm{nil}) \rightrightarrows (R, \mathrm{nil})} \qquad \frac{(R, D) \rightrightarrows (R', D')}{(R, (t{+}1, \mathtt{sr}, w) :: D) \rightrightarrows (R', (t, \mathtt{sr}, w) :: D')}$$

$$\frac{(R, D) \rightrightarrows (R', D') \qquad \mathtt{sr} \in \mathrm{dom}(R)}{(R, (0, \mathtt{sr}, w) :: D) \rightrightarrows (R'\{\mathtt{sr} \rightsquigarrow w\}, D')}$$

$$\frac{(R, D) \rightrightarrows (R', D') \qquad \mathtt{sr} \notin \mathrm{dom}(R)}{(R, (0, \mathtt{sr}, w) :: D) \rightrightarrows (R', D')}$$

Note that the write of `sr` has no effect if `sr` is not in the domain of R. Since R is defined as a partial map, we can prove the following lemma.

Lemma 1. *$(R, D) \rightrightarrows (R', D')$ and $R = R_1 \uplus R_2$, if and only if there exists R'_1 and R'_2, such that $(R_1, D) \rightrightarrows (R'_1, D')$, $(R_2, D) \rightrightarrows (R'_2, D')$, and $R' = R'_1 \uplus R'_2$.*

Here the disjoint union $R_1 \uplus R_2$ represents the union of R_1 and R_2 if they have disjoint domains, and undefined otherwise. This lemma is important to give sound semantics to delay buffer related assertions, as discussed in Sec. 3.

The transition steps for individual instructions are classified into three categories: the control transfer steps ($\_ \vdash \_ \circ\!\!\longrightarrow \_$), the steps for `save`, `restore` and `wr` instructions ($\_ \bullet\!\!\longrightarrow \_$), and the steps for other simple instructions ($\_ \longrightarrow \_$). The corresponding step transition relations are defined inductively in Fig. 7 (b), (c) and (d) respectively.

Note that, after the control-transfer instructions, `pc` is set to `npc` and `npc` contains the target code pointer. This explains the one cycle delay for the control transfer. The `call` instruction saves `pc` into the register $\mathtt{r}_{15}$, while `retl` uses $\mathtt{r}_{15}+8$ as the return address (which is the address for the second instruction following the `call`). The conditional branch `be f` jumps to `f` (after one-cycle delay) if the value in the register `z` is *not* 0. Evaluation of expressions `a` and `o` are defined in Fig. 7 (e). Here, we define the sum of two values v_1 and v_2 below. The result of $v_1{+}v_2$ is legal, if both of the v_1 and v_2 are words (Int32), or v_1 is an address and v_2 is an offset. The offset is a word, which acts as an immediate value in the calculation of address.

$$v_1 + v_2 \ ::= \begin{cases} w_1 + w_2 & \textit{if } v_1 = w_1, \text{ and } v_2 = w_2 \\ (b, w_1 + w_2) & \textit{if } v_1 = (b, w_1), \text{ and } v_2 = w_2 \\ \bot & \text{otherwise} \end{cases}$$

The `wr` wants to save the bitwise exclusive OR of the operands into the special register `sr`, but it puts the write into the delay buffer D instead of updating R immediately. The operation $\mathbf{set\_delay}(\mathtt{sr}, w, D)$ is defined below:

$$\mathbf{set\_delay}(\mathtt{sr}, w, D) ::= (X, \mathtt{sr}, w) :: D$$

where X ($0 \le X \le 3$) is a predefined system parameter for the delay cycle.

$$\frac{(R,D) \rightrightarrows (R',D') \quad C \vdash ((M,(R',F),D'),\mathtt{pc},\mathtt{npc}) \circ\!\!\longrightarrow ((M',(R'',F'),D''),\mathtt{pc}',\mathtt{npc}')}{(C,(M,(R,F),D),\mathtt{pc},\mathtt{npc}) ::\!\Longrightarrow (C,(M',(R'',F'),D''),\mathtt{pc}',\mathtt{npc}')}$$

(a) Program Transition

$$\frac{C(\mathtt{pc}) = \mathtt{i} \qquad (M,(R,F),D) \bullet\!\xrightarrow{\mathtt{i}} (M',(R',F'),D')}{C \vdash ((M,(R,F),D),\mathtt{pc},\mathtt{npc}) \circ\!\!\longrightarrow ((M',(R',F'),D'),\mathtt{npc},\mathtt{npc}+4)}$$

$$\frac{C(\mathtt{pc}) = \mathtt{jmp\ a} \qquad [\![\mathtt{a}]\!]_R = \mathtt{f}}{C \vdash ((M,(R,F),D),\mathtt{pc},\mathtt{npc}) \circ\!\!\longrightarrow ((M,(R,F),D),\mathtt{npc},\mathtt{f})}$$

$$\frac{C(\mathtt{pc}) = \mathtt{call\ f} \qquad \mathtt{r}_{15} \in \mathrm{dom}(R)}{C \vdash ((M,(R,F),D),\mathtt{pc},\mathtt{npc}) \circ\!\!\longrightarrow ((M,(R\{\mathtt{r}_{15} \rightsquigarrow \mathtt{pc}\},F),D),\mathrm{npc},\mathtt{f})}$$

$$\frac{C(\mathtt{pc}) = \mathtt{retl} \quad R(\mathtt{r}_{15}) = \mathtt{f}}{C \vdash ((M,(R,F),D),\mathtt{pc},\mathtt{npc}) \circ\!\!\longrightarrow ((M,(R,F),D),\mathtt{npc},\mathtt{f}+8)}$$

$$\frac{C(\mathtt{pc}) = \mathtt{be\ f} \quad R(\mathtt{z}) \neq 0}{C \vdash ((M,(R,F),D),\mathtt{pc},\mathtt{npc}) \circ\!\!\longrightarrow ((M,(R,F),D),\mathtt{npc},\mathtt{f})} \qquad \frac{C(\mathtt{pc}) = \mathtt{be\ f} \quad R(\mathtt{z}) = 0}{C \vdash ((M,(R,F),D),\mathtt{pc},\mathtt{npc}) \circ\!\!\longrightarrow ((M,(R,F),D),\mathtt{npc},\mathtt{npc}+4)}$$

(b) Control Transfer Instruction Transition

$$\frac{(M,R) \xrightarrow{\mathtt{i}} (M',R')}{(M,(R,F),D) \bullet\!\xrightarrow{\mathtt{i}} (M',(R',F),D)} \qquad \frac{\begin{array}{l}R(\mathtt{r}_s) = w_1 \quad [\![\mathtt{o}]\!]_R = w_2 \quad w = w_1 \oplus w_2 \\ \mathtt{sr} \in \mathrm{dom}(R) \quad D' = \mathbf{set\_delay}(\mathtt{sr},w,D)\end{array}}{(M,(R,F),D) \bullet\!\xrightarrow{\mathtt{wr\ r}_s\ \mathtt{o\ sr}} (M,(R,F),D')}$$

$$\frac{\mathbf{save}(R,F) = (R',F') \quad [\![\mathtt{o}]\!]_R = v \quad v' = R(\mathtt{r}_s)+v}{(M,(R,F),D) \bullet\!\xrightarrow{\mathtt{save\ r}_s\ \mathtt{o\ r}_d} (M,(R\{\mathtt{r}_d \rightsquigarrow v'\},F'),D)}$$

$$\frac{\mathbf{restore}(R,F) = (R',F') \quad [\![\mathtt{o}]\!]_R = v \quad v' = R(\mathtt{r}_s)+v}{(M,(R,F),D) \bullet\!\xrightarrow{\mathtt{restore\ r}_s\ \mathtt{o\ r}_d} (M,(R\{\mathtt{r}_s \rightsquigarrow v'\},F'),D)}$$

(c) Save, Restore and Wr instruction Transition

$$\frac{R(\mathtt{sr}) = w \qquad \mathtt{r}_d \in \mathrm{dom}(R)}{(M,R) \xrightarrow{\mathtt{rd\ sr\ r}_d} (M,R\{\mathtt{r}_d \rightsquigarrow w\})} \qquad \frac{R(\mathtt{r}_s) = v_1 \quad [\![\mathtt{o}]\!]_R = v_2 \quad v = v_1+v_2 \quad \mathtt{r}_d \in \mathrm{dom}(R)}{(M,R) \xrightarrow{\mathtt{add\ r}_s\ \mathtt{o\ r}_d} (M,R\{\mathtt{r}_d \rightsquigarrow v\})}$$

$$\frac{[\![\mathtt{a}]\!]_R = l \quad M(l) = v' \quad \mathtt{r}_d \in \mathrm{dom}(R)}{(M,R) \xrightarrow{\mathtt{ld\ a\ r}_d} (M,R\{\mathtt{r}_d \rightsquigarrow v'\})}$$

(d) Simple Instruction Transition

$$[\![\mathtt{o}]\!]_R ::= \begin{cases} R(r) & \textit{if } \mathtt{o} = r \\ w & \textit{if } \mathtt{o} = w, \\ & \quad -4096 \leq w \leq 4095 \\ \bot & \text{otherwise} \end{cases} \qquad [\![\mathtt{a}]\!]_R ::= \begin{cases} [\![\mathtt{o}]\!]_R & \textit{if } \mathtt{a} = \mathtt{o} \\ v_1+v_2 & \textit{if } \mathtt{a} = \mathtt{r}+\mathtt{o},\ R(\mathtt{r}) = v_1 \\ & \quad \textbf{and } [\![\mathtt{o}]\!]_R = v_2 \\ \bot & \text{otherwise} \end{cases}$$

(e) Expression Semantics

Fig.7. Selected operational semantics rules (taken from [10])

The `save` and `restore` instructions rotate the register windows and update the register file. Their operations over F and R are defined in Fig. 6.

3 Program Logic

In this section, we use a simplified version of our program logic that doesn't support refinement verification to present how our logic handles the features of SPARCv8. The relational program logic for refinement reasoning will be introduced in Sec. 4.

3.1 Assertions

$$\begin{aligned}(Asrt)\ p,q ::= {} & \mathsf{emp} \mid l \mapsto v \mid \mathtt{rn} \mapsto v \mid \rhd_t \mathtt{sr} \mapsto w \mid p\downarrow \\ & \mid \mathtt{cwp} \mapsto (\!| w_{id}, F |\!) \mid p \wedge q \mid p \vee q \mid p * q \\ & \mid \mathtt{a} =_a v \mid \mathtt{o} = v \mid \forall x.\, p \mid \exists x.\, q \mid \ldots\end{aligned}$$

Fig.8. Syntax of Assertions

We define the syntax of assertions ($Asrt$) in Fig. 8, and their semantics in Fig. 9. We extend separation logic assertions with specifications of delay buffers and register windows. Registers are like variables in separation logic, but are treated as resources. The assertion emp says that the memory and the register file are both empty. $l \mapsto v$ specifies a singleton memory cell with value v stored in the address l. $\mathtt{rn} \mapsto v$ says that $\mathtt{rn}$ is the only register in the register file and it contains the value v. Also $\mathtt{rn}$ is *not* in the delay buffer. Separating conjunction $p * q$ has the standard semantics as in separation logic [11].

The assertion $\rhd_t \mathtt{sr} \mapsto w$ describes a delayed write in the delay buffer D. It describes the uncertainty of $\mathtt{sr}$'s value in R, which is unknown for now but will become w in up to $t+1$ cycles. We use $\_ \rightrightarrows^k \_$ to represent k-step execution of the delayed writes in D. It also requires that there be at most one delayed write for a specific special register $\mathtt{sr}$ in D (*i.e.* $\mathbf{noDup}(\mathtt{sr}, D)$). This prevents more than one delayed write to the same register within 4 instruction cycles, which practically have no restrictions on programming. By the semantics we have (the $p \Rightarrow q$ means for any S, if $S \models p$ holds, then $S \models q$ holds)

$$\mathtt{sr} \mapsto w \ \Rightarrow\ \rhd_t \mathtt{sr} \mapsto w \qquad \rhd_t \mathtt{sr} \mapsto w \ \Rightarrow\ \rhd_{t+k} \mathtt{sr} \mapsto w$$

The assertion $p\downarrow$ allows us to reduce the uncertainty by executing one step of the delayed writes. It specifies states reachable after executing one step of delayed writes from those states satisfying p. Therefore we know:

$$(\rhd_0 \mathtt{sr} \mapsto w)\downarrow \Rightarrow\ \mathtt{sr} \mapsto w \qquad (\rhd_{t+1} \mathtt{sr} \mapsto w)\downarrow \Rightarrow\ \rhd_t \mathtt{sr} \mapsto w$$

Also it is easy to see that if p syntactically does not contain sub-terms in the form of $\rhd_t \mathtt{sr} \mapsto w$, then $(p\downarrow) \Leftrightarrow p$.

The following lemma shows $(\_)\downarrow$ is distributive over separating conjunction.

Lemma 2. $(p * q)\downarrow \Leftrightarrow (p\downarrow) * (q\downarrow)$.

The lemma can be proved following Lemma 1. Let

$$\begin{aligned} R_1 \uplus R_2 ::= {} & R_1 \cup R_2 \qquad \textit{if } R_1 \perp R_2 \\ M_1 \uplus M_2 ::= {} & M_1 \cup M_2 \qquad \textit{if } M_1 \perp M_2 \end{aligned}$$

and we present the proof of Lemma 2 below.

Proof. "⇒": if $(M, (R, F), D) \models (p * q)\downarrow$, then $(M, (R, F), D) \models (p)\downarrow * (q)\downarrow$. From the semantics of assertions, we get there exists R' and D', such that:

$$(R', D') \rightrightarrows (R, D) \tag{2}$$

$$(M, (R', F), D') \models p * q \tag{3}$$

From (3), there exists M_1, M_2, R'_1 and R'_2 such that:

$$M = M_1 \uplus M_2 \tag{4}$$

$$R' = R'_1 \uplus R'_2 \tag{5}$$

$$(M_1, (R'_1, F), D') \models p \tag{6}$$

$$(M_2, (R'_2, F), D') \models q \tag{7}$$

By applying Lemma 1 on (2) and (5), we get that there exists R_1 and R_1, where $R = R_1 \uplus R_2$, such that

$$(R'_1, D') \rightrightarrows (R_1, D) \tag{8}$$

$$(R'_2, D') \rightrightarrows (R_2, D) \tag{9}$$

From (6) and (8), we get $(M_1, (R_1, F), D) \models p\downarrow$; and from (7) and (9), we get $(M_2, (R_2, F), D) \models q\downarrow$. Thus, we prove that $(M, (R, F), D) \models (p\downarrow) * (q\downarrow)$ holds.

$$
\begin{array}{ll}
S \models \mathtt{emp} & ::= S.M = \emptyset \wedge S.Q.R = \emptyset \\
S \models l \mapsto v & ::= S.M = \{l \rightsquigarrow v\} \wedge S.Q.R = \emptyset \\
S \models \mathtt{rn} \mapsto v & ::= S.Q.R = \{\mathtt{rn} \rightsquigarrow v\} \wedge \mathtt{rn} \notin \mathrm{dom}(S.D) \wedge S.M = \emptyset \\
S \models \triangleright_t \mathtt{sr} \mapsto w & ::= \exists k, R', D'.\, 0 \le k \le t{+}1 \wedge (R, D) \rightrightarrows^k (R', D') \wedge \\
 & \qquad ((M, (R', F), D') \models \mathtt{sr} \mapsto w) \wedge \mathbf{noDup}(D, \mathtt{sr}) \\
 & \qquad \text{where } S = (M, (R, F), D) \\
S \models p\downarrow & ::= \exists R', D'.\ ((M, (R', F), D') \models p) \wedge (R', D') \rightrightarrows (R, D) \\
 & \qquad \text{where } S = (M, (R, F), D) \\
S \models \mathtt{cwp} \mapsto (\!| w_{id}, F |\!) & ::= (S \models \mathtt{cwp} \mapsto w_{id}) \wedge \exists F'.\, F \cdot F' = S.Q.F \\
S \models \mathtt{a} =_a v & ::= [\![\mathtt{a}]\!]_{S.Q.R} = v \wedge \mathbf{word\_align}(v) \\
S \models \mathtt{o} = v & ::= [\![\mathtt{o}]\!]_{S.Q.R} = v \\
S \models p_1 * p_2 & ::= \exists S_1, S_2.\, S_1 \models p_1 \wedge S_2 \models p_2 \wedge S = S_1 \uplus S_2
\end{array}
$$

$$
M_1 \perp M_2 \ ::= (\mathrm{dom}(M_1) \cap \mathrm{dom}(M_2)) = \emptyset \qquad R_1 \perp R_2 \ ::= (\mathrm{dom}(R_1) \cap \mathrm{dom}(R_2)) = \emptyset
$$

$$
S_1 \uplus S_2 \ ::= \begin{cases} (M_1 \cup M_2, (R_1 \cup R_2, F), D) & \text{if } M_1 \perp M_2 \wedge R_1 \perp R_2 \wedge \\ & \quad S_1 = (M_1, (R_1, F), D) \wedge S_2 = (M_2, (R_2, F), D) \\ \textbf{undefined} & \text{otherwise} \end{cases}
$$

$$
\mathrm{dom}(D) \ ::= \begin{cases} \{\mathtt{sr}\} \cup \mathrm{dom}(D') & \textbf{if } D = (t, \mathtt{sr}, w) :: D' \\ \emptyset & \text{if } D = \mathrm{nil} \end{cases}
$$

$$
\mathbf{noDup}(D, \mathtt{sr}) \ ::= \begin{cases} \mathtt{sr} \notin \mathrm{dom}(D') & \text{if } D = (t, \mathtt{sr}, w) :: D' \\ \mathtt{sr} \neq \mathtt{sr}' \wedge \mathbf{noDup}(D', \mathtt{sr}) & \text{if } D = (t, \mathtt{sr}', w) :: D' \\ \textbf{True} & \text{if } D = \mathrm{nil} \end{cases}
$$

Fig.9. Semantics of Assertions

"$\Leftarrow$": if $(M, (R, F), D) \models (p\downarrow) * (q\downarrow)$, then $(M, (R, F), D) \models (p * q)\downarrow$. From the semantics of assertions, we get there exists M_1, M_2, R'_1, R'_2, R_1, R_2, D'_1 and D'_2, where $R = R_1 \uplus R_2$, such that:

$$(R'_1, D'_1) \rightrightarrows (R_1, D),\ (R'_2, D'_2) \rightrightarrows (R_2, D) \tag{10}$$

$$(M_1, (R'_1, F), D'_1) \models p \tag{11}$$

$$(M_2, (R'_2, F), D'_2) \models q \tag{12}$$

By the definition of the step of the delayed writes, we can prove that $D'_1 = D'_2$. Let $D' = D'_1 = D'_2$. By applying Lemma 1 on (10), we get there exists R', such that $R' = R'_1 \uplus R'_2$ and the following holds:

$$(R', D') \rightrightarrows (R, D) \tag{13}$$

From (11), (12) and (2), we prove that $(M, (R, F), D) \models (p * q)\downarrow$ holds. Thus, we are done. □

We use $\mathtt{cwp} \mapsto (\!| w_{id}, F |\!)$ to describe the pointer cwp of the current register window and the frame list as a circular stack. Note that F is just a prefix of the frame list, since usually we do not need to know contents of the full list. Here we use $F \cdot F'$ to represent the concatenation of lists F and F'. Therefore we have $\mathtt{cwp} \mapsto (\!| w_{id}, F \cdot F' |\!) \implies \mathtt{cwp} \mapsto (\!| w_{id}, F |\!)$.

The assertions $\mathtt{a} =_a v$ and $\mathtt{o} = v$ describe the value of a and o respectively. They are intuitionistic assertions. Since a is used as an address, we also require it to be properly aligned on a 4-byte boundary. We define **word_align** to represent this restriction below. The result of the address expression a may be a word, if it is a pointer in code heap, or a memory address, if it is a location of memory.

$$\mathbf{word\_align}(v) ::= \exists w.\, (v = w \vee v = (\_, w)) \wedge w\%4 = 0$$

3.2 Inference Rules

The code specification θ and code heap specification Ψ are defined below:

(valList) $\iota \in$ list value (pAsrt) $\text{fp}, \text{fq} \in \text{valList} \to Asrt$
(CdSpec) $\theta ::= (\text{fp}, \text{fq})$ (CdHpSpec) $\Psi ::= \{\mathtt{f} \rightsquigarrow \theta\}^*$

The code heap specification Ψ maps the code labels for basic blocks to their specifications θ, which is a pair of pre- and post-conditions. Instead of using normal assertions, the pre- and post-conditions are assertions parameterized over a list of values ι. They play the role of auxiliary variables — Feeding the pre- and the post-conditions with the same ι allows us to establish relationship of states specified in the pre- and post-conditions.

Although we assign a specification θ to each basic block, the post-condition does not specify the states reached at the end of the block. Instead, it specifies the condition that needs to be specified in the future when the *current function* returns. This follows the idea developed in SCAP [7], but we use the standard unary state assertion instead of the binary state assertions used in SCAP, so that existing proof techniques (such as Coq tactics) for standard Hoare-triples can be applied to simplify the verification process.

– $\{(\text{fp}, \text{fq})\}$
```
add  %i0, %i1, %l7
add  %l7, %i2, %l7
retl
nop
```

$$\begin{aligned}\text{fp} ::= \lambda\, lv.\, &((\%\mathtt{i}_0 \mapsto lv[0]) * (\%\mathtt{i}_1 \mapsto lv[1]) * (\%\mathtt{i}_2 \mapsto lv[2])\\ &* \%\mathtt{l}_7 \mapsto \_ * (\mathtt{r}_{15} \mapsto lv[3]))\\ &\wedge (lv[1], lv[2], lv[3] \in \text{Word})\end{aligned}$$

$$\begin{aligned}\text{fq} ::= \lambda\, lv.\, &(\%\mathtt{i}_0 \mapsto lv[0]) * (\%\mathtt{i}_1 \mapsto lv[1]) * (\%\mathtt{i}_2 \mapsto lv[2])\\ &* (\%\mathtt{l}_7 \mapsto lv[0] + lv[1] + lv[2]) * (\mathtt{r}_{15} \mapsto lv[3])\end{aligned}$$

Fig.10. Example for Function Specification

We give a simple example in Fig. 10 to show a specification for a function, which simply sums the values of the registers $\%\mathtt{i}_0$, $\%\mathtt{i}_1$ and $\%\mathtt{i}_2$ and writes the result into the register $\%\mathtt{l}_7$. The specification (fp, fq) says that, when provided with the same lv as argument, the function preserves the value of $\%\mathtt{i}_0$, $\%\mathtt{i}_1$ and $\%\mathtt{i}_2$, $\%\mathtt{l}_7$ in the beginning contains any value and at the end contains the sum of $\%\mathtt{i}_0$, $\%\mathtt{i}_1$ and $\%\mathtt{i}_2$, and the function also preserves the value of $\mathtt{r}_{15}$, which it uses as the return address. To verify the function, we need to prove that it satisfies $(\text{fp}\, lv, \text{fq}\, lv)$ for all lv. Here, $lv[1]$ and $lv[2]$ cannot be a memory address, because a value plus a memory address is illegal. $lv[3]$ also should be a word, because it is a return code pointer whose type is word.

Fig. 11 shows selected inference rules in our logic. Our logic divides the proof work into three layers. We define the well-formed code heap in the form of $(\vdash C : \Psi)$ to verify the code heap C, the well-formed instruction sequence in the form of $(\Psi \vdash \{(p, q)\}\ \mathtt{f} : \mathbb{I})$ to verify the instruction sequence $\mathbb{I}$ starting from $\mathtt{f}$ in code heap, and well-formed instruction in the form of $(\vdash \{p\}\, \mathtt{i}\, \{q\})$ to verify the single simple instruction $\mathtt{i}$.

The top rule **CDHP** verifies the code heap C. It requires that every basic block specified in Ψ can be verified with respect to the specification, with any argument ι used to instantiate the pre- and post-conditions.

The **SEQ** rule is applied when meeting an instruction sequence starting with a simple instruction $\mathtt{i}$. The instruction $\mathtt{i}$ is verified by the corresponding well-formed instruction rules, with the precondition $p{\downarrow}$ and some post-condition p'. We use $p{\downarrow}$ because there is an implicit step executing delayed writes before executing every instruction. The post-condition p' for $\mathtt{i}$ is then used as the precondition to verify the remaining part of the instruction sequence.

Delayed control transfers. We distinguish the `jmp` and `call` instructions — The former makes an *intra-function* control transfer, while the latter makes func-

$\boxed{\vdash C : \Psi}$ **(Well-Formed Code Heap)**

$$\frac{\text{for all } \mathtt{f} \in \mathrm{dom}(\Psi),\ \iota :\ \ \Psi(\mathtt{f}) = (\mathrm{fp}, \mathrm{fq}) \quad \Psi \vdash \{(\mathrm{fp}\ \iota, \mathrm{fq}\ \iota)\}\ \mathtt{f} : C[\mathtt{f}]}{\vdash C : \Psi}\ \textbf{(CDHP)}$$

$\boxed{\Psi \vdash \{(p,q)\}\ \mathtt{f} : \mathbb{I}}$ **(Well-Formed Instruction Sequences)**

$$\frac{\vdash \{p{\downarrow}\}\ \mathtt{i}\ \{p'\} \qquad \Psi \vdash \{(p', q)\}\ \mathtt{f}{+}4 : \mathbb{I}}{\Psi \vdash \{(p,q)\}\ \mathtt{f} :\ \mathtt{i};\ \mathbb{I}}\ \textbf{(SEQ)}$$

$$\frac{\begin{array}{c} p{\downarrow} \Rightarrow\ (\mathtt{a} {=_a} \mathtt{f}') \quad \mathtt{f}' \in \mathrm{dom}(\Psi) \quad \Psi(\mathtt{f}') = (\mathrm{fp}, \mathrm{fq}) \\ \vdash \{p{\downarrow}{\downarrow}\}\ \mathtt{i}\ \{p'\} \quad \exists \iota, p_r.\ (p' \ \Rightarrow\ \mathrm{fp}\ \iota * p_r) \ \wedge\ (\mathrm{fq}\ \iota * p_r \ \Rightarrow\ q) \end{array}}{\Psi \vdash \{(p,q)\}\ \mathtt{f} :\ \mathtt{jmp\ a};\ \mathtt{i}}\ \textbf{(JMP)}$$

$$\frac{\begin{array}{c} \mathtt{f}' \in \mathrm{dom}(\Psi) \quad \Psi(\mathtt{f}') = (\mathrm{fp}, \mathrm{fq}) \quad \Psi \vdash \{(p', q)\}\ \mathtt{f}{+}8 : \mathbb{I} \\ p{\downarrow} \Rightarrow\ (\mathtt{r}_{15} {\mapsto} \_) * p_1 \quad \vdash \{(\mathtt{r}_{15} {\mapsto} \mathtt{f} * p_1){\downarrow}\}\ \mathtt{i}\ \{p_2\} \\ \exists \iota, p_r.\ (p_2 \Rightarrow \mathrm{fp}\ \iota * p_r) \ \wedge\ (\mathrm{fq}\ \iota * p_r \Rightarrow p') \ \wedge\ (\mathrm{fq}\ \iota \Rightarrow \mathtt{r}_{15} {=} \mathtt{f}) \end{array}}{\Psi \vdash \{(p,q)\}\ \mathtt{f} :\ \mathtt{call\ f'};\ \mathtt{i};\ \mathbb{I}}\ \textbf{(CALL)}$$

$$\frac{p{\downarrow}{\downarrow} \Rightarrow (\mathtt{r}_{15} {\mapsto} \mathtt{f}') * p_1 \quad \vdash \{p_1\}\ \mathtt{i}\ \{p_2\} \quad (\mathtt{r}_{15} {\mapsto} \mathtt{f}') * p_2 \Rightarrow q}{\Psi \vdash \{(p,q)\}\ \mathtt{f} :\ \mathtt{retl};\ \mathtt{i}}\ \textbf{(RETL)}$$

$$\frac{\begin{array}{c} p{\downarrow} \Rightarrow (\mathtt{z} {\mapsto} w) * \mathrm{true} \quad \mathtt{f}' \in \mathrm{dom}(\Psi) \quad \Psi(\mathtt{f}') = (\mathrm{fp}, \mathrm{fq}) \\ \vdash \{p{\downarrow}{\downarrow}\}\ \mathtt{i}\ \{p'\} \quad \Psi \vdash \{(p' \wedge w = 0, q)\}\ \mathtt{f}{+}8 : \mathbb{I} \\ \exists \iota, p_r.\ ((p' \wedge w \neq 0) \Rightarrow \mathrm{fp}\ \iota * p_r) \ \wedge\ (\mathrm{fq}\ \iota * p_r \Rightarrow q) \end{array}}{\Psi \vdash \{(p,q)\}\ \mathtt{f} :\ \mathtt{be\ f'}; \mathtt{i}; \mathbb{I}}\ \textbf{(BE)}$$

$\boxed{\vdash \{p\}\ \mathtt{i}\ \{q\}}$ **(Well-Formed Instructions)**

$$\frac{\mathtt{sr} {\mapsto} \_ * p \Rightarrow (\mathtt{r}_s = w_1 \ \wedge\ \mathtt{o} = w_2)}{\vdash \{\mathtt{sr} {\mapsto} \_ * p\}\ \mathtt{wr\ r}_s\ \mathtt{o\ sr}\ \{(\triangleright_3 \mathtt{sr} {\mapsto} (w_1 \oplus w_2)) * p\}}\ \textbf{(WR)}$$

$$\frac{}{\vdash \{\mathtt{sr} {\mapsto} w * \mathtt{r}_d {\mapsto} \_\}\ \mathtt{rd\ sr\ r}_d\ \{\mathtt{sr} {\mapsto} w * \mathtt{r}_d {\mapsto} w\}}\ \textbf{(RD)}$$

$$\frac{\begin{array}{c} p \Rightarrow (\mathtt{r}_s = v_1 \wedge \mathtt{o} = v_2) \qquad w'_{id} = \mathbf{next\_cwp}(w_{id}) \qquad w\ \&\ 2^{w'_{id}} = 0 \qquad v = v_1 {+} v_2 \\ p \Rightarrow (\mathtt{cwp} {\mapsto} (\!|\, w_{id}, F \bullet \_ \bullet \_ \,|\!)) * (\mathsf{out} {\mapsto} \mathrm{fm}_o) * (\mathsf{local} {\mapsto} \mathrm{fm}_l) * (\mathsf{in} {\mapsto} \mathrm{fm}_i) * p_1 \\ (\mathtt{cwp} {\mapsto} (\!|\, w'_{id}, \mathrm{fm}_l :: \mathrm{fm}_i :: F \,|\!)) * (\mathsf{out} {\mapsto} \_) * (\mathsf{local} {\mapsto} \_) * (\mathsf{in} {\mapsto} \mathrm{fm}_o) * p_1 \Rightarrow \mathtt{r}_d {\mapsto} \_ * p_2 \end{array}}{\vdash \{(\mathtt{wim} {\mapsto} w) * p\}\ \mathtt{save\ r}_s\ \mathtt{o\ r}_d\ \{(\mathtt{wim} {\mapsto} w) * (\mathtt{r}_d {\mapsto} v) * p_2\}}\ \textbf{(SAVE)}$$

where $[\mathtt{r}_i, \ldots, \mathtt{r}_{i+7}] {\mapsto} [w_0, \ldots, w_7]\ ::= \mathtt{r}_i {\mapsto} w_0 * \cdots * \mathtt{r}_{i+7} {\mapsto} w_7$
and out, local and in are defined in Fig. 6.

$$\frac{\begin{array}{c} p \Rightarrow (\mathtt{r}_s = v_1 \wedge \mathtt{o} = v_2) \qquad w'_{id} = \mathbf{prev\_cwp}(w_{id}) \qquad w\ \&\ 2^{w'_{id}} = 0 \qquad v = v_1 {+} v_2 \\ p \Rightarrow (\mathtt{cwp} {\mapsto} (\!|\, w_{id}, \mathrm{fm}_1 :: \mathrm{fm}_2 :: F \,|\!)) * (\mathsf{out} {\mapsto} \_) * (\mathsf{local} {\mapsto} \_) * (\mathsf{in} {\mapsto} \mathrm{fm}_i) * p_1 \\ (\mathtt{cwp} {\mapsto} (\!|\, w'_{id}, F \bullet \_ \bullet \_ \,|\!)) * (\mathsf{out} {\mapsto} \mathrm{fm}_i) * (\mathsf{local} {\mapsto} \mathrm{fm}_1) * (\mathsf{in} {\mapsto} \mathrm{fm}_2) * p_1 \Rightarrow \mathtt{r}_d {\mapsto} \_ * p_2 \end{array}}{\vdash \{(\mathtt{wim} {\mapsto} w) * p\}\ \mathtt{restore\ r}_s\ \mathtt{o\ r}_d\ \{(\mathtt{wim} {\mapsto} w) * (\mathtt{r}_d {\mapsto} v) * p_2\}}\ \textbf{(RESTORE)}$$

Fig.11. Seleted Inference Rules

tion calls. The **JMP** rule requires that the target address is a valid one specified in Ψ. Starting from the precondition p, after executing the instruction `i` following **JMP** and the corresponding delayed writes, the post-condition p' of `i` should satisfy the precondition of the target instruction sequence, with some instantiation ι of the logical variables and a frame assertion p_r. Since the target instruction sequence of `jmp` is in the same function as the `jmp` instruction itself, the post-condition fq specified at the target address (with the same instantiation ι of the logical variables and the frame assertion p_r) should meet the post-condition q of the current function. As we explained before, the post-condition q does not specify the states reached at the end of the instruction sequence (which are specified by p' instead).

The **CALL** rule is similar to the **JMP** rule in that it also requires the post-condition p_2 of the instruction `i` following the `call` satisfy the precondition of the target instruction sequence, with some instantiation ι of the logical variables and a frame assertion p_r. Here we need to record that the code label `f` is saved in $\mathtt{r}_{15}$ by the `call` instruction. When the callee returns, its post-condition fq (with the same instantiation of auxiliary variables ι) needs to ensure $\mathtt{r}_{15}$ still contains `f`, so that the callee returns to the correct address. Also the fq with the frame p_r needs to satisfy the precondition p' for the remaining instruction sequences of the caller.

The **RETL** rule simply requires that the post-condition q holds at the end of the instruction `i` following `retl`. Also `i` cannot touch the register $\mathtt{r}_{15}$, therefore $\mathtt{r}_{15}$ specified in p must be the same as in q. Since at the calling point we already required that the post-condition of the callee guarantees $\mathtt{r}_{15}$ contains the correct return address, we know $\mathtt{r}_{15}$ contains the correct value before `retl`.

The **BE** rule checks the *current* value of register `z` and decides whether the branch will be taken after executing the following instruction `i`. If the value of `z` is not zero, the branch is taken and this rule does the same things as the **JMP** rule; otherwise, the branch is *not* taken and the remaining instruction sequence $\mathbb{I}$ should be well-formed.

Delayed writes and register windows. The bottom layer of our logic is for well-formed instructions. The **WR** rule requires the ownership of the target register `sr` in the precondition ($\mathtt{sr} \mapsto \_$). Also it implies there is no delayed writes to `sr` in the delay buffer (see the semantics defined in Fig. 9). At the end of the delayed write, we use $\rhd_3 \mathtt{sr} \mapsto w_1 \oplus w_2$ to indicate the new value will be ready in up to 3 cycles. Since the maximum delay cycle X cannot be bigger than 3 and the value of X may vary in different systems, programmers usually take a conservative approach to assume $X = 3$ for portability of code. Our rule reflects this conservative view. The **RD** rule says the special register can be read only if it is not in the delay buffer. The **SAVE** and **RESTORE** rules reflect the save and recovery of the execution contexts, which is consistent with the operational semantics of the `save` and `restore` instructions given in Figs. 6 and 7.

4 Refinement Verification of SPARCv8

In this section, we present our *relational* program logic for refinement verification of SPARCv8 code. As an extension of our conference paper, it consists of the following work:

- We define a new Pseudo-SPARCv8 language as the high-level specification language in Sec. 4.1;
- We make some modifications to the SPARCv8 language defined in Sec. 2 and let it act as the low-level language in our refinement verification.

We present our low-level SPARCv8 language and the modifications in Sec. 4.2;

- We define the correctness of the abstract assembly primitives in Sec. 4.3, which is formulated as *contextual refinement* between the implementations and their corresponding abstract assembly primitives;
- We present a new program logic to do refinement verification in Sec. 4.4;
- We show that our new program logic is sound in Sec. 4.5. The semantics of judgements, different from the previous work in our conference paper which only ensures the partial correctness of verified programs, are defined as simulation relations between the low- and high-level programs which guarantees contextual refinement.

4.1 High-level Pseudo-SPARCv8 Language

The Pseudo-SPARCv8 language contains two parts: the SPARCv8 code as client and the set of abstract assembly primitives. Here, we require that the execution of client SPARCv8 code preserves a restriction between register window and stack in memory, shown on the left side of Fig. 12 (`cwp` points to the current window and `wim` marks the invalid window, the details of overlapping of adjacent windows are omitted in the figure).

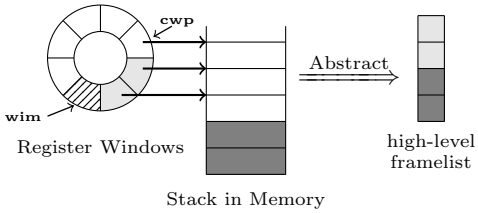


Fig.12. Abstraction of context management

During the execution of SPARCv8 program, part of previous procedures' contexts (the light gray part in the left side of Fig. 12) are saved in register window, the others (the dark gray part in the left side of Fig. 12) are stored in stack in memory, because the number of windows is limited. The restriction is that the stack pointer (%`sp`) of each procedure, including the current and previous ones, whose context is saved in register window currently, should point to the top of its stack frame (shown as the thick arrow in Fig. 12), so that the contexts in these windows can be stored correctly in memory when needed. For instance, the context switch routine will check whether the previous window is valid (in clockwise direction in Fig. 12), and use instruction `restore` to set it as the current one and save its contents into stack (in memory) until the previous one is invalid (filled with east north lines in Fig. 12). We require that the execution of client code preserves such restriction. Otherwise, some SPARCv8 functions like context switch routine, whose executions will store the contexts saved in register windows into stack in memory, cannot be verified if it is unclear where to save the contents of some windows. We do the following when defining Pseudo-SPARCv8 program to make the execution of client code preserve such restriction:

- In order to ensure that the stack pointer (%`sp`) always points to the top of its stack frame, we require that each instruction, like `add` and `ld`, whose execution doesn't operate register window, is prohibited to update the value of %`sp`; and as for the `save` and `restore`, we require them to be used in specific forms. We introduce "`Psave` w" as a macro of "`save` %`sp`, $-w$, %`sp`", whose execution generates a new %`sp` pointing to the stack frame size w allocated newly for the next window. We also introduce "`Prestore`" as a macro of "`restore` $\%g_0, \%g_0, \%g_0$" [1], whose execution just restores the previous window and doesn't modify

[1] In SPARCv8, $\%g_0$ is always 0, and usually used as parameters when instructions do not require specific parameters.

the value of any register in the previous window restored. The original `save` and `restore` instructions have *no* semantics in high-level client code.

- The special registers in SPARCv8 usually play specific roles and modifying them should be done carefully. For example, `wim` marks which window is invalid.

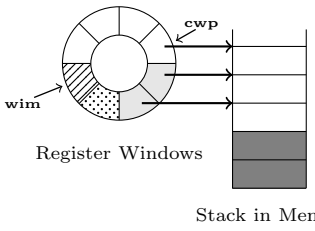


Fig.13. Problem of modifying `wim` arbitrary

If we change its value shown in Fig. 12 to mark another window invalid, as shown in Fig. 13, and call context switch routine to save the contents of previous windows into memory until the invalid one at this moment, a problem will arise since we don't know where to save the contents of window marked invalid originally (filled with dots in Fig. 13). So, we forbid the client to modify special registers and give *no* semantics to instruction `wr` in high-level client code. Modifying them is hidden in the implementations of the abstract assembly primitives in low-level. And the delay buffer can be omitted in high-level program state.

- As shown in Fig. 12, we find that we can abstract the register window and memory in stack storing contexts into a list (defined as HFrmList formally in Fig. 15). After this abstraction, we don't need to care about whether the contexts are saved in register windows or memory, and don't need to describe the contents of windows unused (the windows in white color in Fig. 12, but excluding the current one pointed by `cwp`) in the Pseudo-SPARCv8 level. The `cwp` register is no longer needed in Pseudo-SPARCv8 program because the register window is abstracted away. The low-level program in our work doesn't use this abstraction, because the low-level program should be realistically modelled, and the implementations of some primitives need to know the existance of register windows, for instance, the context switch routine needs to save the contents of the register window into stack (in memory).

We define the syntax of the high-level Pseudo-SPARCv8 language in Fig. 14. The code (HCode) Π has two parts: the code heap C and the set of abstract primitives (PrimSet) Ω, which is a partial mapping from labels to abstract assembly primitive. The code heap C in Π acts as the client to call abstract assembly primitives. The abstract assembly primitive (Prim) Υ is defined as a relation that takes a list of values as arguments and maps a high-level program state (defined in Fig. 15) to another. We add three pseudo instructions in simple instruction (SimpIns). The `Psave` w and `Prestore` restrict the instruction `save` and `restore` to be used in the specific form as mentioned before. We also introduce `print r`, whose execution outputs the value v in general register `r` and generates an message `out`(v) as an observable event. The high-level message (HMsg) α can be either an empty message τ, or an output `out`(v), or a `call`$(\mathtt{f}, \overline{v})$ meaning to call a primitive labelled `f` with arguments $\overline{v}$.

The machine states (HState) of the high-level Pseudo-SPARCv8 program are defined in Fig. 15. The high-level program $\mathbb{P}$ is a pair of high-level code Π and high-level state $\mathbb{S}$. High-level state is a tuple including: a thread pool (ThrdPool) T, current thread id (Tid) `t`, the thread local state (ThrdLcSt) $\mathcal{K}$ of the current thread, and the memory (Mem) M.

$$
\begin{array}{rrcl}
\text{(HCode)} & \Pi & ::= & (C, \Omega) \\
\text{(PrimSet)} & \Omega & ::= & \{\mathtt{f}_1 \rightsquigarrow \Upsilon_1, \ldots, \mathtt{f}_n \rightsquigarrow \Upsilon_n\} \\
\text{(Comm)} & c & ::= & \mathtt{i} \mid \mathtt{call\ f} \mid \mathtt{jmp\ a} \mid \mathtt{retl} \mid \mathtt{be\ f} \\
\text{(SimpIns)} & \mathtt{i} & ::= & \mathtt{Psave}\ w \mid \mathtt{Prestore} \mid \mathtt{print\ r} \mid \mathtt{ld\ a\ r}_d \mid \mathtt{st\ r}_s\ \mathtt{a} \mid \mathtt{add\ r}_s\ \mathtt{o\ r}_d \mid \mathtt{rd\ sr\ r}_d \mid \mathtt{wr\ r}_s\ \mathtt{o\ sr} \\
& & & \mid \mathtt{save\ r}_s\ \mathtt{o\ r}_d \mid \mathtt{restore\ r}_s\ \mathtt{o\ r}_d \mid \ldots \\
\text{(HMsg)} & \alpha & ::= & \tau \mid \mathsf{out}(v) \mid \mathsf{call}(\mathtt{f}, \overline{v})
\end{array}
\qquad
\begin{array}{rrcl}
\text{(CodeHeap)} & C & \in & \text{Word} \rightharpoonup \text{Comm} \\
\text{(Prim)} & \Upsilon & \in & \text{List Val} \to \text{HState} \to \text{HState} \to \text{Prop}
\end{array}
$$

Fig.14. Syntax of Pseudo-SPARCv8 Code

$$
\begin{array}{rrcl}
\text{(HProg)} & \mathbb{P} & ::= & (\Pi, \mathbb{S}) \\
\text{(Tid)} & \mathtt{t} & \in & \mathbb{Z} \\
\text{(HRegFile)} & \mathbb{R} & \in & \text{HRegName} \to \text{Val} \\
\text{(HFrmList)} & \mathbb{F} & ::= & \text{nil} \mid (\text{fm}_1, \text{fm}_2) :: \mathbb{F}
\end{array}
\qquad
\begin{array}{rrcl}
\text{(HState)} & \mathbb{S} & ::= & (T, \mathtt{t}, \mathcal{K}, M) \\
\text{(ThrdLcSt)} & \mathcal{K} & ::= & (\mathbb{Q}, \mathtt{pc}, \mathtt{npc}) \\
\text{(HRegName)} & \hat{\mathtt{rn}} & ::= & \mathtt{r}_0 \mid \ldots \mid \mathtt{r}_{31} \mid \mathtt{n} \mid \mathtt{z} \mid \mathtt{c} \mid \mathtt{v} \\
\text{(HFrame)} & \text{fm} & ::= & [v_0, \ldots, v_7]
\end{array}
\qquad
\begin{array}{rrcl}
\text{(ThrdPool)} & T & ::= & \{\mathtt{t} \rightsquigarrow \mathcal{K}\}^* \\
\text{(HRstate)} & \mathbb{Q} & ::= & (\mathbb{R}, \mathbb{F})
\end{array}
$$

Fig.15. Machine States for Pseudo-SPARCv8 Code

Thread Local State. The thread local state $\mathcal{K}$ is a triple of high-level register state (HRstate) $\mathbb{Q}$, and program counters `pc` and `npc`. The high-level register state $\mathbb{Q}$ consists of the high-level register file (HRegFile) $\mathbb{R}$, and the high-level frame list (HFrmList) $\mathbb{F}$. $\hat{\mathtt{rn}}$ is the high-level register names (HRegName), where the `cwp` is omitted as introduced before and we also omit special registers for simplicity, because we forbid the high-level client code to modify them [2]. The high-level frame list $\mathbb{F}$ is a list of pairs $(\text{fm}_1, \text{fm}_2)$, which is used to save the contexts (local and in registers) fm_1 and fm_2 of previous procedures. After introducing the state of high-level program, we define the primitive switch as an instantiation of Υ below:

$$
\begin{array}{l}
\mathsf{switch} ::= \\
\quad \lambda\, \overline{v}, \mathbb{S}, \mathbb{S}'.\ \exists \mathtt{t}'.\ M(\mathsf{TaskNew}) = (\mathtt{t}', 0) \wedge T(\mathtt{t}') = (\mathbb{Q}', \mathtt{pc}', \mathtt{npc}') \\
\qquad\quad \wedge\, T' = T\{\mathtt{t} \rightsquigarrow (\mathbb{Q}, \mathtt{pc}, \mathtt{npc})\} \wedge \mathtt{t} \neq \mathtt{t}' \wedge \overline{v} = \text{nil} \\
\quad \text{where } \mathbb{S} = (T, \mathtt{t}, (\mathbb{Q}, \mathtt{pc}, \mathtt{npc}), M), \\
\qquad\quad \mathbb{S}' = (T', \mathtt{t}', (\mathbb{Q}', \mathtt{f}+8, \mathtt{f}+12), M), \mathtt{f} = \mathbb{Q}'.\mathbb{R}(\mathtt{r}_{15}).
\end{array}
$$

The execution of the switch primitive takes no arguments ($\overline{v} = \text{nil}$), and changes the identifier of the current thread according to the value in location TaskNew. We use parameters $\mathbb{S}$ and $\mathbb{S}'$ to represent the machine states before and after execution of switch respectively.

Operational Semantics in High-level. The operational semantics for high-level Pseudo-SPARCv8 program is defined in Fig. 16. The high-level program transition relation $(\Pi, \mathbb{S}) :\xRightarrow{\alpha} (\Pi, \mathbb{S}')$ is defined in Fig. 16 (a). In each step, the program may either execute the instruction pointed by `pc` and generate empty message τ or an output $\mathsf{out}(v)$, or call an abstract assembly primitive in the primitive set. When calling an abstract assembly primitive, the execution of current thread (defined as $(\_ \Vdash \_ \circ\xrightarrow{=} \_)$ in Fig. 16 (b)) will generate a message $\mathsf{call}(\mathtt{f}, \overline{v})$, which means that it hopes to call the abstract assembly primitive Υ labelled `f`, which is *not* in the domain of code heap C, with arguments $\overline{v}$ (we use $\mathbf{args}(\mathbb{Q}, M, \overline{v})$ to get arguments $\overline{v}$ from high-level state, and its definition is omitted here).

The thread local step is defined in Fig. 16 (b). Here, the step for simple instruction `i` is represented as "$\mathbf{exec}(\mathtt{i}, \_, \_)$". We show the state transition relation for pseudo instructions `Psave` w and `Prestore` in Fig. 16 (c). Supposing the current register state $\mathbb{Q}$ is $(\mathbb{R}, \mathbb{F})$, executing instruction `Psave` w will save the local and in registers into high-level frame list $\mathbb{F}$. It also allocates a new block b of size 64 byte to w

[2]There is no problem to reserve special registers in high-level register file and permit the high-level client code to read them.

$$\frac{\Pi = (C, \Omega) \qquad C \Vdash (\mathcal{K}, M) \circ\!\!\xrightarrow{\tau} (\mathcal{K}', M')}{(\Pi, (T, \mathtt{t}, \mathcal{K}, M)) :\!\!\Longrightarrow (\Pi, (T, \mathtt{t}, \mathcal{K}', M'))} \qquad \frac{\Pi = (C, \Omega) \qquad C \Vdash (\mathcal{K}, M) \circ\!\!\xrightarrow{\mathsf{out}(v)} (\mathcal{K}', M)}{(\Pi, (T, \mathtt{t}, \mathcal{K}, M)) :\!\!\xRightarrow{\mathsf{out}(v)} (\Pi, (T, \mathtt{t}, \mathcal{K}', M))}$$

$$\frac{\begin{array}{c}\Pi = (C, \Omega) \qquad C \Vdash (\mathcal{K}, M) \circ\!\!\xrightarrow{\mathsf{call}(\mathtt{f}, \overline{v})} (\mathcal{K}', M) \qquad \Omega(\mathtt{f}) = \Upsilon \\ \Upsilon\,(\overline{v})(T, \mathtt{t}, \mathcal{K}', M)(T', \mathtt{t}', \mathcal{K}'', M')\end{array}}{(\Pi, (T, \mathtt{t}, \mathcal{K}, M)) :\!\!\Longrightarrow (\Pi, (T', \mathtt{t}', \mathcal{K}'', M'))}$$

(a) High-level Program Transition

$$\frac{C(\mathtt{pc}) = \mathtt{i} \qquad \mathbf{exec}(\mathtt{i}, (\mathbb{Q}, M), (\mathbb{Q}', M'))}{C \Vdash ((\mathbb{Q}, \mathtt{pc}, \mathtt{npc}), M) \circ\!\!\xrightarrow{\tau} ((\mathbb{Q}', \mathtt{npc}, \mathtt{npc} + 4), M')}$$

$$\frac{C(\mathtt{pc}) = \mathtt{call\ f} \quad \mathtt{r}_{15} \in \mathrm{dom}(\mathbb{R})}{C \Vdash (((\mathbb{R}, \mathbb{F}), \mathtt{pc}, \mathtt{npc}), M) \circ\!\!\xrightarrow{\tau} (((\mathbb{R}\{\mathtt{r}_{15} \rightsquigarrow \mathtt{pc}\}, \mathbb{F}), \mathtt{npc}, \mathtt{f}), M)}$$

$$\frac{C(\mathtt{pc}) = \mathtt{retl} \qquad \mathbb{R}(\mathtt{r}_{15}) = \mathtt{f}}{C \Vdash (((\mathbb{R}, \mathbb{F}), \mathtt{pc}, \mathtt{npc}), M) \circ\!\!\xrightarrow{\tau} (((\mathbb{R}, \mathbb{F}), \mathtt{npc}, \mathtt{f} + 8), M)}$$

$$\frac{C(\mathtt{pc}) = \mathtt{print\ r} \qquad \mathbb{R}(\mathtt{r}) = v}{C \Vdash (((\mathbb{R}, \mathbb{F}), \mathtt{pc}, \mathtt{npc}), M) \circ\!\!\xrightarrow{\mathsf{out}(v)} (((\mathbb{R}, \mathbb{F}), \mathtt{npc}, \mathtt{npc} + 4), M)}$$

$$\frac{\mathtt{pc} \notin \mathrm{dom}(C) \qquad \mathtt{npc} = \mathtt{pc} + 4 \qquad \mathbf{args}(\mathbb{Q}, M, \overline{v})}{C \Vdash ((\mathbb{Q}, \mathtt{pc}, \mathtt{npc}), M) \circ\!\!\xrightarrow{\mathsf{call}(\mathtt{pc}, \overline{v})} ((\mathbb{Q}, \mathtt{pc}, \mathtt{npc}), M)}$$

(b) High-level Thread Local Transition

$$\frac{\begin{array}{c}\mathbb{Q} = (\mathbb{R}, \mathbb{F}) \qquad \mathbb{R}' = \mathbb{R}\{\mathsf{out} \rightsquigarrow \_, \mathsf{local} \rightsquigarrow \_, \mathsf{in} \rightsquigarrow \mathbb{R}([\mathsf{out}])\}\{\%\mathtt{sp} \rightsquigarrow (b, 0)\} \\ \boldsymbol{alloc}(M, b, 64, w) = M' \qquad \mathbb{Q}' = (\mathbb{R}', (\mathbb{R}([\mathsf{local}]), \mathbb{R}([\mathsf{in}])) :: \mathbb{F})\end{array}}{\mathbf{exec}(\mathtt{Psave}\ w, (\mathbb{Q}, M), (\mathbb{Q}', M'))}$$

$$\frac{\begin{array}{c}\mathbb{Q} = (\mathbb{R}, (\mathrm{fm}_1, \mathrm{fm}_2) :: \mathbb{F}) \qquad \mathbb{R}(\%\mathtt{sp}) = (b, 0) \qquad \boldsymbol{free}(b, M) = M' \\ \mathbb{R}' = \mathbb{R}\{\mathsf{out} \rightsquigarrow \mathbb{R}([\mathsf{in}]), \mathsf{local} \rightsquigarrow \mathrm{fm}_1, \mathsf{in} \rightsquigarrow \mathrm{fm}_2\} \qquad \mathbb{Q}' = (\mathbb{R}', \mathbb{F})\end{array}}{\mathbf{exec}(\mathtt{Prestore}, (\mathbb{Q}, M), (\mathbb{Q}', M'))}$$

(c) High-level Instruction Transition

Fig.16. Selected operational semantics rules for high-level program

byte as a new stack frame in memory (represented as $\boldsymbol{alloc}(M, b, 64, w) = M'$). The reason why it starts from 64 byte is that the 0 to 64 bytes (16 words) in a stack frame are usually reserved to save the context in window (local and in registers) by convention [5], and this part of memory is abstracted away in Pseudo-SPARCv8 program as we have explained and shown in Fig. 12. The instruction `Prestore` does the reverse, freeing the block of current stack frame (represented as $\boldsymbol{free}(b, M) = M'$), and restoring the context of the previous procedure saved in $\mathbb{F}$.

4.2 Low-level SPARCv8 Program

The global program transition of the low-level SPARCv8 program is defined as the following form:

$$\frac{\begin{array}{l}(R, D) \rightrightarrows (R', D') \\ C \vdash ((M, (R', F), D'), \mathtt{pc}, \mathtt{npc}) \circ\!\!\xrightarrow{\tau/\mathsf{out}(v)} ((M', (R'', F'), D''), \mathtt{pc}', \mathtt{npc}')\end{array}}{(C, (M, (R, F), D), \mathtt{pc}, \mathtt{npc}) ::\!\!\xRightarrow{\tau/\mathsf{out}(v)} (C, (M', (R'', F'), D''), \mathtt{pc}', \mathtt{npc}')}$$

The low-level SPARCv8 program is slightly different from the SPARCv8 program defined in Sec. 2:

1. The low-level SPARCv8 program uses the instruc-

tions defined in Fig. 14. It means that we need to give semantics to the pseudo instructions `Psave`, `Prestore` and `print` in the low-level SPARCv8 program. Since the `Psave` and `Prestore` are simply special forms of `save` and `restore` as explained, and `print` is a primitive responsible for generating observable events, defining their semantics is not a challenge and the translation of programs in this modified language into ones in the standard SPARCv8 language is trivial.

2. The program transition defined in Sec. 2 does not generate observable events. Here, in order to support refinement verification, since we use the event trace refinement [12], each step of the program generates either an empty message τ, or an output $\mathsf{out}(v)$ produced by `print`.

Note that the client code and the implementations of abstract assembly primitives in low-level are both SPARCv8 code heap. So, we do not need to define their linking in semantics. More details of the low-level program can be found in our Coq code and TR [8].

4.3 Primitive Correctness

We first establish a state relation between low- and high-level program states. We define this relation below ($\uplus$ means disjoint union), shown as "$S \sim \mathbb{S}$".

$$\frac{M = M_c \uplus M_T \uplus \{\mathsf{TaskCur} \rightsquigarrow (\mathtt{t}, 0)\} \uplus M' \quad (M_c, Q) \Downarrow_{\mathsf{c}} (\mathtt{t}, \mathcal{K}) \quad M_T \Downarrow_r T\backslash\{\mathtt{t}\} \quad D = \mathrm{nil}}{(M, Q, D) \sim (T, \mathtt{t}, \mathcal{K}, M')}$$

The low-level memory M is split into four parts: M_c used to save the context of the current thread `t`; M_T saving the contexts of the ready threads, except the current thread `t`; a singleton memory cell located $\mathsf{TaskCur}$ saving the current thread id; and shared memory M' that is same as the high-level memory. The delayed buffer D is nil, because client is not permitted to modify any special register through instructions `wr`. The memory M_T used to save the contexts of the ready threads is *abstracted* as a thread pool in high-level program. Their relation is represented as "$M_T \Downarrow_{\mathsf{r}} T\backslash\{\mathtt{t}\}$". We use "$(M_c, Q) \Downarrow_{\mathsf{c}} (\mathtt{t}, \mathcal{K})$" to represent the state relation of current thread `t` in low- and high-level programs.

The correctness of abstract assembly primitives is defined in terms of *contextual refinement*. We give its formal definition in Def. 1. And we use *event trace refinement* proposed by Liang *et al.* [12].

Definition 1 (Primitive Correctness). *$C_{\mathrm{as}} \sqsubseteq \Omega$ iff for any C, S, $\mathbb{S}$, `pc` and `npc`, if $S \sim \mathbb{S}$, and $\mathsf{ProgSafe}(\mathbb{P})$, then $\mathrm{P} \subseteq \mathbb{P}$ holds. (where $\mathrm{P} = (C \uplus C_{\mathrm{as}}, S, \mathtt{pc}, \mathtt{npc})$, $\mathbb{P} = ((C, \Omega), \mathbb{S})$, and $\mathbb{S}.\mathcal{K} = (\_, \mathtt{pc}, \mathtt{npc})$).*

We use a code heap C_{as} to represent the implementations and Ω to represent the set of corresponding abstract assembly primitives. The contextual refinement, denoted as $C_{\mathrm{as}} \sqsubseteq \Omega$, says that if and only if for any client code (or context) C, low-level program state S, high-level program state $\mathbb{S}$, program counters `pc` and `npc`, if the low- and high-level program states satisfy the state relation $S \sim \mathbb{S}$ and the high-level program will never get stuck (shown as $\mathsf{ProgSafe}(\mathbb{P})$), then there is an *event trace refinement* [12], which means that P produces no more observable behaviors than $\mathbb{P}$ and is denoted as $\mathrm{P} \subseteq \mathbb{P}$, between low- and high-level programs. $\mathsf{ProgSafe}(\mathbb{P})$ is defined formally below:

$$\mathsf{ProgSafe}(\mathbb{P}) \ ::= \forall\, \mathbb{P}'.\, (\mathbb{P} :\Longrightarrow^* \mathbb{P}') \Longrightarrow (\exists\, \mathbb{P}''.\, \mathbb{P}' :\Longrightarrow \mathbb{P}'')$$

The client code C is SPARCv8 code, so the high-level code is a pair of C and Ω, and the low-level code is just the union of C and C_{as}, shown as $(C \uplus C_{\mathrm{as}})$, because both of C and C_{as} are SPARCv8 code heap:

$$C \uplus C_{\mathrm{as}} \ ::= C \cup C_{\mathrm{as}} \quad \textit{if } \mathrm{dom}(C) \cap \mathrm{dom}(C_{\mathrm{as}}) = \emptyset$$

$$
\begin{array}{lll}
(S,\mathbb{S},A,w)\models \texttt{Emp} & ::= & \mathbb{S}.M=\emptyset \wedge \mathbb{S}.T=\emptyset \wedge S\models \mathsf{emp} \\
(S,\mathbb{S},A,w)\models p & ::= & S\models p \wedge \mathbb{S}.M=\emptyset \wedge \mathbb{S}.T=\emptyset \\
(S,\mathbb{S},A,w)\models \hat{\texttt{rn}} \rightarrowtail v & ::= & \mathbb{S}.\mathcal{K}.\mathbb{Q}.\mathbb{R}(\hat{\texttt{rn}})=v \wedge (S,\mathbb{S},A,w)\models \texttt{Emp} \\
(S,\mathbb{S},A,w)\models l \rightarrowtail v & ::= & \mathbb{S}.M=\{l\rightsquigarrow v\} \wedge \mathbb{S}.T=\emptyset \wedge S\models \mathsf{emp} \\
(S,\mathbb{S},A,w)\models \mathtt{t}\rightsquigarrow_{\mathsf{c}}\mathcal{K} & ::= & \mathbb{S}.T\backslash\{\mathtt{t}\}=\emptyset \wedge \mathbb{S}.\mathtt{t}=\mathtt{t} \wedge \mathbb{S}.\mathcal{K}=\mathcal{K} \wedge \mathbb{S}.M=\emptyset \wedge S\models \mathsf{emp} \\
(S,\mathbb{S},A,w)\models \mathtt{t}\rightsquigarrow_{\mathsf{r}}\mathcal{K} & ::= & \mathbb{S}.T=\{\mathtt{t}\rightsquigarrow\mathcal{K}\} \wedge \mathbb{S}.M=\emptyset \wedge \mathtt{t}\neq\mathbb{S}.\mathtt{t} \wedge S\models \mathsf{emp} \\
(S,\mathbb{S},A,w)\models (\!|A'|\!) & ::= & A=A' \wedge (S,\mathbb{S},A,w)\models \texttt{Emp} \\
(S,\mathbb{S},A,w)\models \blacklozenge(w') & ::= & w'\leq w \wedge (S,\mathbb{S},A,w)\models \texttt{Emp} \\
(S,\mathbb{S},A,w)\models \mathtt{p}\downarrow & ::= & \exists S'.((S',\mathbb{S},A,w)\models \mathtt{p}) \wedge (R',D')\rightrightarrows(R,D) \\
 & & \quad \text{where } S=(M,(R,F),D),\ S'=(M,(R',F),D') \\
(S,\mathbb{S},A,w)\models \mathtt{p}*\mathtt{q} & ::= & \exists S_1,S_2,\mathbb{S}_1,\mathbb{S}_2,w_1,w_2.\ S=S_1\uplus S_2 \wedge \mathbb{S}=\mathbb{S}_1\uplus\mathbb{S}_2 \wedge \\
 & & \quad w=w_1+w_2 \wedge (S_1,\mathbb{S}_1,A,w_1)\models \mathtt{p} \wedge (S_2,\mathbb{S}_2,A,w_2)\models \mathtt{q}
\end{array}
$$

$$
T_1 \perp T_2 \ ::= (\mathrm{dom}(T_1)\cap\mathrm{dom}(T_2))=\emptyset
$$

$$
\mathbb{S}_1\uplus\mathbb{S}_2 \ ::= \begin{cases} (T_1\cup T_2,\mathtt{t},\mathcal{K},M_1\cup M_2) & \textit{if } T_1\perp T_2 \wedge M_1\perp M_2 \wedge \\ & \quad \mathbb{S}_1=(T_1,\mathtt{t},\mathcal{K},M_1)\wedge\mathbb{S}_2=(T_2,\mathtt{t},\mathcal{K},M_2) \\ \textbf{undefined} & \text{otherwise} \end{cases}
$$

$$
\text{(HPrimCom)}\quad A ::= \Upsilon(\overline{v}) \mid \perp \qquad \frac{\Upsilon\,(\overline{v})(\mathbb{S})(\mathbb{S}')}{(\Upsilon(\overline{v}),\mathbb{S})\dashrightarrow(\perp,\mathbb{S}')}
$$

Fig.18. Semantics of Relation Assertion

4.4 Relational Program Logic for Refinement Verification

$$
\begin{array}{lll}
\textit{(RelAsrt)} & \mathtt{p},\mathtt{q} ::= & p \mid \hat{\texttt{rn}}\rightarrowtail v \mid l\rightarrowtail v \mid \texttt{Emp} \\
 & & \mid \mathtt{t}\rightsquigarrow_{\mathsf{c}}\mathcal{K} \mid \mathtt{t}\rightsquigarrow_{\mathsf{r}}\mathcal{K} \mid (\!|A|\!) \mid \blacklozenge(w) \\
 & & \mid \mathtt{p}\downarrow \mid \mathtt{p}\wedge\mathtt{q} \mid \mathtt{p}\vee\mathtt{q} \mid \mathtt{p}*\mathtt{q} \mid \ldots
\end{array}
$$

Fig.17. Syntax of Relational Assertion

Relational Assertion Fig. 17 gives the *relational* assertion language, and its semantics is given in Fig. 18. The relational assertions are interpreted over relational state $(S,\mathbb{S},A,w)$, which contains the low-level state S, the high-level state $\mathbb{S}$, the abstract assembly primitive command A defined in Fig. 18, and a word w recording the number of the tokens. The high-level primitive command A is either an abstract assembly primitive Υ parameterized with its arguments $\overline{v}$, or a $\perp$ meaning the primitive has already been executed. The relational assertion $\mathtt{p}$ reserves original assertion p describing the low-level state S.

We define $\hat{\texttt{rn}}\rightarrowtail v$ and $l\rightarrowtail v$ to describe the state of register file and memory in high-level. The assertion $\texttt{Emp}$ says that the high-level memory and thread pool are both empty, and the low-level state satisfies emp defined in Fig. 9. The assertion $\mathtt{t}\rightsquigarrow_{\mathsf{c}}\mathcal{K}$ and $\mathtt{t}\rightsquigarrow_{\mathsf{r}}\mathcal{K}$ represent the thread local state of current and ready thread respectively. Note that the threads in thread pool are viewed as resources and can be separated by separation conjunction.

The assertion $(\!|A|\!)$ means the current high-level primitive command is A. And the assertion $\blacklozenge(w)$ takes a word w, which can also be separated by separation conjunction, to state that the number of tokens in current state is *no less* than w. In the introduction of the inference rules following, we use tokens to avoid infinite loops and recursive calls to make sure the termination preserving refinement.

The assertion $\mathtt{p}\downarrow$ describes the state after executing one step of the delayed writes. Suppose a rela-

tional state $(S', \mathbb{S}, A, w)$ satisfies the assertion $\mathtt{p}$, where $S' = (M, (R', F), D')$, then $\mathtt{p}\downarrow$ holds after executing a step of delayed writes $((R', F') \rightrightarrows (R, F))$ from S'.

Inference Rules in Relational Program Logic

The code specification $\hat{\theta}$ and code heap specification Ψ for refinement verification are defined below :

(valList) $\iota \in$ list value (pAsrt) $\mathtt{fp}, \mathtt{fq} \in$ valList $\rightarrow$ *RelAsrt*
(CdSpec) $\hat{\theta} ::= (\mathtt{fp}, \mathtt{fq})$ (CdHpSpec) $\Psi ::= \{\mathtt{f} \rightsquigarrow \hat{\theta}\}^*$

Here, $\mathtt{fp}$ and $\mathtt{fq}$ are relational assertions parameterized over a list of values ι. Fig. 19 shows selected inference rules for refinement verification in our logic. The top rule **WfPrim** verifies the contextual refinement between the code heap C_{as} and the corresponding abstract assembly primitive set Ω. It requires that each code block specified in Ψ can be verified with respect to its specification, shown as $(\vdash C_{\text{as}} : \Psi)$, and the specification of the implementation of the abstract assembly primitive needs to meet some restrictions, shown as $\mathsf{wdSpec}(\mathtt{fp}, \mathtt{fq}, \Upsilon)$, which we will discuss in more details following. The inference rules for `jmp` and `call` in relational program logic will consume a token, shown as $\blacklozenge(1)$, in order to avoid infinite loops and recursive function calls, because our refinement relation is termination-preserving. The $\mathsf{wf}(\mathtt{p}_r)$, whose definition is omitted here, means there is no sub-term in form of $(\mathtt{t} \rightsquigarrow_{\mathtt{c}} \mathcal{K})$, $(\hat{\mathtt{rn}} \rightarrowtail v)$ and $(\!|A|\!)$ in frame $\mathtt{p}_r$, because the state they described is not separated by separation conjunction $*$. The **ABSCSQ** rule allows us to execute the high-level primitive command specified in precondition. The implication $\mathtt{p} \Rrightarrow \mathtt{p}'$ is defined below formally:

$$(\mathtt{p} \Rightarrow \mathtt{p}') \vee (\forall S, \mathbb{S}, A, w.\, ((S, \mathbb{S}, A, w) \models \mathtt{p}) \Longrightarrow (\exists \mathbb{S}', A', w'.\, ((A, \mathbb{S}) \dashrightarrow (A', \mathbb{S}')) \wedge ((S, \mathbb{S}', A', w') \models \mathtt{p}'))$$

The inference rules for verifying instructions are not presented here, because they are no different from the rules shown in Fig. 11.

Well-defined Specification. The $\mathsf{wdSpec}(\mathtt{fp}, \mathtt{fq}, \Upsilon)$ is defined formally in Def. 2. It contains three properties that the specifications need to satisfy, and we explain them in turn in the following.

Definition 2 (Well-defined Specification)**.** $\mathsf{wdSpec}(\mathtt{fp}, \mathtt{fq}, \Upsilon)$ *holds, iff*

1. *for any* $\overline{v}$, $\mathbb{S}$, $\mathbb{S}'$, $\mathbb{S}_r$. *if* $\Upsilon(\overline{v})(\mathbb{S})(\mathbb{S}')$, *and* $\mathbb{S} \perp \mathbb{S}_r$, *then the following holds (where* $\mathtt{f} = \mathbb{S}'.\mathcal{K}.\mathbb{Q}.\mathbb{R}(\mathtt{r}_{15})$*):*
   - $\mathbb{S}'.\mathcal{K}.\mathtt{pc} = \mathtt{f}+8$, $\mathbb{S}'.\mathcal{K}.\mathtt{npc} = \mathtt{f}+12$*;*
   - *there exists* $\mathbb{S}''$, $\mathbb{S}'_r$, $\Upsilon(\overline{v})(\mathbb{S} \uplus \mathbb{S}_r)(\mathbb{S}'')$, $\mathbb{S}'' = \mathbb{S}' \uplus \mathbb{S}'_r$, *and* $\mathbb{S}_r.T = \mathbb{S}'_r.T$, $\mathbb{S}_r.M = \mathbb{S}'_r.M$*;*
2. *for any* ι, *there exists* $\overline{v}$, *such that* $\mathtt{fp}\ \iota \Rightarrow (\!|\Upsilon(\overline{v})|\!) * \text{true}$, *and* $\mathtt{fq}\ \iota \Rightarrow (\!|\bot|\!) * \text{true}$*;*
3. *for any* $\overline{v}, S, \mathbb{S}$, *if* $(S, \mathbb{S}, \_, \_) \in \mathsf{INV}(\Upsilon(\overline{v}), \overline{v})$, *there exists* $\iota, \mathtt{p}_r$ *and* w, *such that* $(S, \mathbb{S}, \Upsilon(\overline{v}), w) \models (\mathtt{fp}\ \iota * \mathtt{p}_r)$, $(\mathtt{fq}\ \iota * \mathtt{p}_r) \Rightarrow \mathsf{INV}(\bot, \_)$, *and* $\mathsf{Sta}(\Upsilon(\overline{v}), \mathtt{p}_r)$ *hold.*

First, the program counters should be equal to $\mathtt{f}+8$ and $\mathtt{f}+12$, where $\mathtt{f}$ is contained in $\mathtt{r}_{15}$ register after the execution of abstract assembly primitive. It ensures that the low-level implementation and high-level abstract assembly primitive will return to the same code pointers after executions. We also require that if an abstract assembly primitive can execute safely on a part of program state, it can also execute safely on the whole program state, and additional program state remains unchanged. Here, $\mathbb{S} \perp \mathbb{S}_r$ is defined formally below:

$$\mathbb{S} \perp \mathbb{S}_r ::= T \perp T' \wedge M \perp M' \wedge \mathtt{t} = \mathtt{t}' \wedge \mathcal{K} = \mathcal{K}' \quad \text{where } \mathbb{S} = (T, \mathtt{t}, \mathcal{K}, M),\ \mathbb{S}_r = (T', \mathtt{t}', \mathcal{K}', M')$$

Second, the abstract assembly primitive should be specified in the precondition, and its execution should be done in the final state. **Third**, there is an *invariant* between low- and high-level programs, holding at the entry of the function. Our logic needs to ensure that such invariant can be reestablished at the exit of function. Such invariant is defined as INV formally below:

$$\mathsf{INV}(A, \overline{v}) ::= \{(S, \mathbb{S}, A, w) \mid S \sim \mathbb{S} \wedge (\exists \mathbb{S}'.\, (A, \mathbb{S}) \dashrightarrow^* (\bot, \mathbb{S}')) \wedge \mathbf{args}(\mathbb{S}.\mathcal{K}.\mathbb{Q},\ \mathbb{S}.M, \overline{v})\}$$

The invariant consists of the state relation between low- and high-level program states, shown as $S \sim \mathbb{S}$, and the safe execution of the primitive command, which means that the primitive command A can execute zero

$\boxed{\Psi \vdash C_{\text{as}} : \Psi}$ **(Well-formed Primitive)**

$$\frac{\begin{array}{l}\vdash C_{\text{as}} : \Psi \\ \text{for all } \mathtt{f} \in \text{dom}(\Omega) : \\ \quad \Psi(\mathtt{f}) = (\mathtt{f}\mathtt{p}, \mathtt{f}\mathtt{q}) \quad \Omega(\mathtt{f}) = \Upsilon \quad \mathsf{wdSpec}(\mathtt{f}\mathtt{p}, \mathtt{f}\mathtt{q}, \Upsilon)\end{array}}{\Psi \vdash C_{\text{as}} : \Omega}\ \textbf{(WfPrim)}$$

$\boxed{\vdash C_{\text{as}} : \Psi}$ **(Well-formed Code Heap)**

$$\frac{\text{for all } \mathtt{f} \in \text{dom}(\Psi),\ \iota :\ \Psi(\mathtt{f}) = (\mathtt{f}\mathtt{p}, \mathtt{f}\mathtt{q}) \quad \Psi \vdash \{(\mathtt{f}\mathtt{p}\ \iota, \mathtt{f}\mathtt{q}\ \iota)\}\,\mathtt{f} : C_{\text{as}}[\mathtt{f}]}{\vdash C_{\text{as}} : \Psi}\ \textbf{(WfInt)}$$

$\boxed{\Psi \vdash \{(\mathtt{p}, \mathtt{q})\}\,\mathtt{f} : \mathbb{I}}$ **(Well-formed Instruction Sequences)**

$$\frac{\Vdash \{\mathtt{p}\downarrow\}\,\mathtt{i}\,\{\mathtt{p}'\} \quad \Psi \vdash \{(\mathtt{p}', \mathtt{q})\}\,\mathtt{f}+4 : \mathbb{I}}{\Psi \vdash \{(\mathtt{p}, \mathtt{q})\}\,\mathtt{f} : \mathtt{i}; \mathbb{I}}\ \textbf{(SEQ)}$$

$$\frac{\begin{array}{c}\mathtt{p}\downarrow \Rightarrow (\mathtt{a} =_a \mathtt{f}') \quad \mathtt{f}' \in \text{dom}(\Psi) \quad \Psi(\mathtt{f}') = (\mathtt{f}\mathtt{p}, \mathtt{f}\mathtt{q}) \\ \Vdash \{\mathtt{p}\downarrow\downarrow\}\,\mathtt{i}\,\{\mathtt{p}' * \blacklozenge(1)\} \quad \exists \iota, \mathtt{p}_r.\ (\mathtt{p}' \Rightarrow \mathtt{f}\mathtt{p}\ \iota * \mathtt{p}_r) \wedge (\mathtt{f}\mathtt{q}\ \iota * \mathtt{p}_r \Rightarrow \mathtt{q}) \wedge \mathsf{wf}(\mathtt{p}_r)\end{array}}{\Psi \vdash \{(\mathtt{p}, \mathtt{q})\}\,\mathtt{f} : \mathtt{jmp}\ \mathtt{a}; \mathtt{i}}\ \textbf{(JMP)}$$

$$\frac{\begin{array}{c}\mathtt{f}' \in \text{dom}(\Psi) \quad \Psi(\mathtt{f}') = (\mathtt{f}\mathtt{p}, \mathtt{f}\mathtt{q}) \quad \Psi \vdash \{(\mathtt{p}', \mathtt{q})\}\,\mathtt{f}+8 : \mathbb{I} \\ \mathtt{p}\downarrow \Rightarrow (\mathtt{r}_{15} \mapsto \_) * \mathtt{p}_1 \quad \Vdash \{(\mathtt{r}_{15} \mapsto \mathtt{f} * \mathtt{p}_1)\downarrow\}\,\mathtt{i}\,\{\mathtt{p}_2 * \blacklozenge(1)\} \\ \exists \iota, \mathtt{p}_r.\ (\mathtt{p}_2 \Rightarrow \mathtt{f}\mathtt{p}\ \iota * \mathtt{p}_r) \wedge (\mathtt{f}\mathtt{q}\ \iota * \mathtt{p}_r \Rightarrow \mathtt{p}') \wedge (\mathtt{f}\mathtt{q}\ \iota \Rightarrow \mathtt{r}_{15} = \mathtt{f}) \wedge \mathsf{wf}(\mathtt{p}_r)\end{array}}{\Psi \vdash \{(\mathtt{p}, \mathtt{q})\}\,\mathtt{f} : \mathtt{call}\ \mathtt{f}'; \mathtt{i}; \mathbb{I}}\ \textbf{(CALL)}$$

$$\frac{\mathtt{p}\downarrow\downarrow \Rightarrow (\mathtt{r}_{15} \mapsto \mathtt{f}') * \mathtt{p}_1 \quad \Vdash \{\mathtt{p}_1\}\,\mathtt{i}\,\{\mathtt{p}_2\} \quad (\mathtt{r}_{15} \mapsto \mathtt{f}') * \mathtt{p}_2 \Rightarrow \mathtt{q}}{\Psi \vdash \{(\mathtt{p}, \mathtt{q})\}\,\mathtt{f} : \mathtt{retl}; \mathtt{i}}\ \textbf{(RETL)}$$

$$\frac{\mathtt{p} \Rrightarrow \mathtt{p}' \quad \Psi \vdash \{(\mathtt{p}', \mathtt{q}')\}\,\mathtt{f} : \mathbb{I} \quad \mathtt{q}' \Rrightarrow \mathtt{q}}{\Psi \vdash \{(\mathtt{p}, \mathtt{q})\}\,\mathtt{f} : \mathbb{I}}\ \textbf{(ABSCSQ)}$$

Fig.19. Selected Inference Rules for Refinement Verification

(if $A = \bot$) or one (if $A = \Upsilon(\overline{v})$) step from the current state, presented as $\exists \mathbb{S}'.\,(A, \mathbb{S})\ \dashrightarrow^*\ (\bot, \mathbb{S}')$, and we use $\dashrightarrow^*$ to denote zero or one step. Including the safe execution of the primitive command is essential because we can get some knowledge of high-level program state from the safe execution of primitive command A. For example, if $\mathsf{INV}(\mathsf{switch}(\text{nil}), \text{nil})$ holds, we know that the location TaskNew must save a pointer pointing to a ready thread in thread pool from the safe execution of primitive switch. Then we in turn know that the memory location TaskNew in low-level state also saves the same pointer from the state relation between low- and high-level program.

$$\begin{array}{l}\mathsf{INV}(\mathsf{switch}(\text{nil}), \text{nil}) \implies \exists \mathsf{t}, \mathcal{K}.\,(\mathsf{t} \rightsquigarrow_{\mathsf{r}} \mathcal{K}) * \\ \qquad (\mathsf{TaskNew} \rightarrowtail (\mathsf{t}, 0)) * (\mathsf{TaskNew} \mapsto (\mathsf{t}, 0)) * \text{true}\end{array}$$

We introduce frame $\mathtt{p}_r$ for local reasoning, and it should be stable under the execution of the abstract assembly primitive (shown as $\mathsf{Sta}(\Upsilon(\overline{v}), \mathtt{p}_r)$ defined below).

$$\begin{array}{l}\mathsf{Sta}(A, \mathtt{p}_r) ::= \forall S_r, \mathbb{S}_r, \mathbb{S}, \mathbb{S}', w. \\ \qquad ((A, \mathbb{S})\ \dashrightarrow^*\ (\bot, \mathbb{S}') \wedge (S_r, \mathbb{S}_r, A, w) \models \mathtt{p}_r \wedge \mathbb{S} \perp \mathbb{S}_r) \\ \qquad\quad \implies \exists \mathbb{S}'_r.\, \mathbb{S}' \perp \mathbb{S}'_r \wedge (S_r, \mathbb{S}'_r, \bot, w) \models \mathtt{p}_r\end{array}$$

4.5 Semantics and Soundness

We first define the simulation relation for instruction sequence. It says C_{as} can execute safely from S, pc and npc until reaching the end of the current instruction sequence ($C_{\text{as}}[\mathtt{pc}]$), and q holds if $C_{\text{as}}[\mathtt{pc}]$ ends

with the return instruction `retl`, and for each step of low-level execution, the high-level program will execute zero or one step. It is formally defined in Def. 3. Here we use "$\_ \longmapsto^n \_$" to represent n-step execution. The w in simulation records the number of tokens. It will be consumed when meeting `jmp` and `call` instructions, so as to avoid infinite loop and recursive function call and ensure termination preserving, and reset when the high-level abstract assembly primitive executes.

Definition 3 (Simulation for Instruction Sequence)**.** $\mathtt{q};\Psi \models (C_{\text{as}}, S, \mathtt{pc}, \mathtt{npc}) \preccurlyeq_w (A, \mathbb{S})$ *holds if and only if the following are true (we omit the case for* `be` *here, which is similar to* `jmp`*):*

1. *if* $C_{\text{as}}(\mathtt{pc}) = \mathtt{i}$ *then:*
   - *there exist* $S', \mathtt{pc}', \mathtt{npc}'$*, such that* $C_{\text{as}} \vdash (S, \mathtt{pc}, \mathtt{npc}) \longmapsto (S', \mathtt{pc}', \mathtt{npc}')$,
   - *for any* S', $\mathtt{pc}'$, $\mathtt{npc}'$, *if* $C_{\text{as}} \vdash (S, \mathtt{pc}, \mathtt{npc}) \longmapsto (S', \mathtt{pc}', \mathtt{npc}')$*, then there exists* $A', \mathbb{S}'$ *and* w'*, such that:*
     (1) *either* $A' = A$, $\mathbb{S}' = \mathbb{S}$ *and* $w' = w$; *or* $(A, \mathbb{S}) \dashrightarrow (A', \mathbb{S}')$,
     (2) $\mathtt{q};\Psi \models (C_{\text{as}}, S', \mathtt{pc}', \mathtt{npc}') \preccurlyeq_{w'} (A', \mathbb{S}')$.
2. *if* $C_{\text{as}}(\mathtt{pc}) = \mathtt{jmp\ a}$ *then:*
   - *there exist* S', $\mathtt{pc}'$, $\mathtt{npc}'$*, such that* $C_{\text{as}} \vdash (S, \mathtt{pc}, \mathtt{npc}) \longmapsto^2 (S', \mathtt{pc}', \mathtt{npc}')$,
   - *for any* S', $\mathtt{pc}'$, $\mathtt{npc}'$, *if* $C_{\text{as}} \vdash (S, \mathtt{pc}, \mathtt{npc}) \longmapsto^2 (S', \mathtt{pc}', \mathtt{npc}')$*, then there exists* $\mathtt{fp}$, $\mathtt{fq}$, ι, A', $\mathbb{S}'$, w', $w'' < w'$ *and* $\mathtt{p}_r$ *such that the following hold:*
     (1) $\mathtt{npc}' = \mathtt{pc}'+4$, $\Psi(\mathtt{pc}') = (\mathtt{fp}, \mathtt{fq})$,
     (2) *either* $A' = A$, $\mathbb{S}' = \mathbb{S}$ *and* $w' = w$; *or* $(A, \mathbb{S}) \dashrightarrow (A', \mathbb{S}')$,
     (3) $(S', \mathbb{S}', A', w'') \models (\mathtt{fp}\ \iota) * \mathtt{p}_r$, $(\mathtt{fq}\ \iota) * \mathtt{p}_r \Rightarrow \mathtt{q}$, $\mathsf{wf}(\mathtt{p}_r)$.
3. *if* $C_{\text{as}}(\mathtt{pc}) = \mathtt{be\ f}$ *then ...*
4. *if* $C_{\text{as}}(\mathtt{pc}) = \mathtt{call\ f}$ *then:*
   - *there exist* S', $\mathtt{pc}'$, $\mathtt{npc}'$*, such that* $C_{\text{as}} \vdash (S, \mathtt{pc}, \mathtt{npc}) \longmapsto^2 (S', \mathtt{pc}', \mathtt{npc}')$,
   - *for any* S', $\mathtt{pc}'$ *and* $\mathtt{npc}'$, *if* $C_{\text{as}} \vdash (S, \mathtt{pc}, \mathtt{npc}) \longmapsto^2 (S', \mathtt{pc}', \mathtt{npc}')$*, then there exist* $\mathtt{fp}, \mathtt{fq}, \iota, A', \mathbb{S}', w', w'' < w'$ *and* $\mathtt{p}_r$*, such that the following hold:*
     (1) $\mathtt{npc}' = \mathtt{pc}'+4$, $\Psi(\mathtt{pc}') = (\mathtt{fp}, \mathtt{fq})$,
     (2) *either* $A' = A$, $\mathbb{S}' = \mathbb{S}$ *and* $w' = w$; *or* $(A, \mathbb{S}) \dashrightarrow (A', \mathbb{S}')$,
     (3) $(S', \mathbb{S}', A', w'') \models (\mathtt{fp}\ \iota) * \mathtt{p}_r$, $\mathsf{wf}(\mathtt{p}_r)$,
     (4) *for any* S_0, $\mathbb{S}_0$, A_0, w_0, *if* $(S_0, \mathbb{S}_0, A_0, w_0) \models (\mathtt{fq}\ \iota) * \mathtt{p}_r$*, then* $\mathtt{q};\Psi \models (C_{\text{as}}, S_0, \mathtt{pc}+8, \mathtt{pc}+12) \preccurlyeq_{w_0} (A_0, \mathbb{S}_0)$,
     (5) $(\mathtt{fq}\ \iota) \Rightarrow (\mathtt{r}_{15} = \mathtt{pc})$.
5. *if* $C_{\text{as}}(\mathtt{pc}) = \mathtt{retl}$ *then :*
   - *there exist* S', $\mathtt{pc}'$, $\mathtt{npc}'$*, such that* $C_{\text{as}} \vdash (S, \mathtt{pc}, \mathtt{npc}) \longmapsto^2 (S', \mathtt{pc}', \mathtt{npc}')$,
   - *for any* S', $\mathtt{pc}'$ *and* $\mathtt{npc}'$, *if* $C_{\text{as}} \vdash (S, \mathtt{pc}, \mathtt{npc}) \longmapsto^2 (S', \mathtt{pc}', \mathtt{npc}')$*, then there exists* A', $\mathbb{S}'$, *and* w', *such that:*
     (1) *either* $A' = A$, $\mathbb{S}' = \mathbb{S}$ *and* $w' = w$; *or* $(A, \mathbb{S}) \dashrightarrow (A', \mathbb{S}')$,
     (2) $(S', \mathbb{S}', A', w') \models \mathtt{q}$, $\mathtt{pc}' = S'.Q.R(\mathtt{r}_{15})+8$, *and* $\mathtt{npc}' = S'.Q.R(\mathtt{r}_{15})+12$.

Then we define the semantics for well-formed instruction sequences and well-formed code heap below.

Definition 4 (Judgment Semantics)**.**

- $\Psi \models \{(\mathtt{p}, \mathtt{q})\}\ \mathtt{f} : \mathbb{I}$ *if and only if, for all* C_{as}, S, $\mathbb{S}$, A *and* w *such that* $C_{\text{as}}[\mathtt{f}] = \mathbb{I}$ *and* $(S, \mathbb{S}, A, w) \models \mathtt{p}$*, we have* $\mathtt{q};\Psi \models (C_{\text{as}}, S, \mathtt{f}, \mathtt{f}+4) \preccurlyeq_w (A, \mathbb{S})$.
- $\models C_{as} : \Psi$ *if and only if, for all* $\mathtt{f}$, $\mathtt{fp}$ *and* $\mathtt{fq}$ *such that* $\Psi(\mathtt{f}) = (\mathtt{fp}, \mathtt{fq})$*, we have* $\Psi \models \{(\mathtt{fp}\ \iota, \mathtt{fq}\ \iota)\}\ \mathtt{f} : C_{\text{as}}[\mathtt{f}]$ *for all* ι.

Next, we define the simulation for function in Def. 5. It means that if there exists a relational state $(S, \mathbb{S}, A, w)$ satisfying the precondition $\mathtt{p}$, then we have the simulation $\mathtt{q} \models (C_{\text{as}}, S, \mathtt{f}, \mathtt{f}+4) \preccurlyeq_i^0 (A, \mathbb{S})$ defined in Def. 6. The definition of simulation $\mathtt{q} \models (C_{\text{as}}, S, \mathtt{pc}, \mathtt{npc}) \preccurlyeq_i^k (A, \mathbb{S})$ carries an index i, which is used to ensure the termination preserving, and the depth k of function call, which increases by the `call` instruction and decreases by `retl` (unless $k = 0$). The simulation relation ensures the safe execution of low-level SPARCv8 function and the corresponding high-level abstract assembly primitive.

Definition 5 (Simulation for Function)**.**

$$(C_{\text{as}}, \mathtt{f}) \preccurlyeq^{(\mathtt{p}, \mathtt{q})} A \ ::= \forall S, \mathbb{S}, w. (S, \mathbb{S}, A, w) \models \mathtt{p} \Longrightarrow \exists i \in \mathit{Index}.\ \mathtt{q} \models (C_{\text{as}}, S, \mathtt{f}, \mathtt{f}+4) \preccurlyeq_i^0 (A, \mathbb{S})$$

where $\mathtt{q} \models (C_{\text{as}}, S, \mathtt{pc}, \mathtt{npc}) \preccurlyeq_i^k (A, \mathbb{S})$ is defined in Def. 6.

Definition 6. $\mathtt{q} \models (C_{\text{as}}, S, \mathtt{pc}, \mathtt{npc}) \preccurlyeq_i^k (A, \mathbb{S})$ *holds if and only if the following are true:*

1. *if* $C_{\text{as}}(\mathtt{pc}) \in \{\mathtt{i}, \mathtt{jmp\ a}, \mathtt{be\ f}\}$, *then:*
   - *there exists* S', $\mathtt{pc}'$, $\mathtt{npc}'$, *such that* $(C_{\text{as}}, S, \mathtt{pc}, \mathtt{npc}) ::\xrightarrow{\tau}: (C_{\text{as}}, S', \mathtt{pc}', \mathtt{npc}')$;
   - *for any* S', $\mathtt{pc}'$, $\mathtt{npc}'$, *if* $(C_{\text{as}}, S, \mathtt{pc}, \mathtt{npc}) ::\xrightarrow{\tau}: (C_{\text{as}}, S', \mathtt{pc}', \mathtt{npc}')$, *then one of the following holds:*
     (a) $\exists\, j < i.\ \mathtt{q} \models (C_{\text{as}}, S', \mathtt{pc}', \mathtt{npc}') \preccurlyeq_j^k (A, \mathbb{S})$;
     (b) *there exists* $\mathbb{S}'$, $j \in$ *Index, such that* $(A, \mathbb{S}) \dashrightarrow (\bot, \mathbb{S}')$ *and* $\mathtt{q} \models (C_{\text{as}}, S', \mathtt{pc}', \mathtt{npc}') \preccurlyeq_j^k (\bot, \mathbb{S}')$ *holds;*
2. *if* $C_{\text{as}}(\mathtt{pc}) = \mathtt{call\ f}$, *then:*
   - *there exists* S', $\mathtt{pc}'$, $\mathtt{npc}'$, *such that* $(C_{\text{as}}, S, \mathtt{pc}, \mathtt{npc}) ::\xrightarrow{\tau}{}^2 (C_{\text{as}}, S', \mathtt{pc}', \mathtt{npc}')$;
   - *for any* S', $\mathtt{pc}'$, $\mathtt{npc}'$, *if* $(C_{\text{as}}, S, \mathtt{pc}, \mathtt{npc}) ::\xrightarrow{\tau}{}^2 (C_{\text{as}}, S', \mathtt{pc}', \mathtt{npc}')$, *then one of the following holds:*
     (a) $\exists\, j < i.\ \mathtt{q} \models (C_{\text{as}}, S', \mathtt{pc}', \mathtt{npc}') \preccurlyeq_j^{k+1} (A, \mathbb{S})$;
     (b) *there exists* $\mathbb{S}'$, $j \in$ *Index, such that* $(A, \mathbb{S}) \dashrightarrow (\bot, \mathbb{S}')$ *and* $\mathtt{q} \models (C_{\text{as}}, S', \mathtt{pc}', \mathtt{npc}') \preccurlyeq_j^{k+1} (\bot, \mathbb{S}')$ *holds;*
3. *if* $C_{\text{as}}(\mathtt{pc}) = \mathtt{retl}$, *then:*
   - *there exists* S', $\mathtt{pc}'$, $\mathtt{npc}'$, *such that* $(C_{\text{as}}, S, \mathtt{pc}, \mathtt{npc}) ::\xrightarrow{\tau}{}^2 (C_{\text{as}}, S', \mathtt{pc}', \mathtt{npc}')$;
   - *for any* S', $\mathtt{pc}'$, $\mathtt{npc}'$, *if* $(C_{\text{as}}, S, \mathtt{pc}, \mathtt{npc}) ::\xrightarrow{\tau}{}^2 (C_{\text{as}}, S', \mathtt{pc}', \mathtt{npc}')$, *then there exists* $j \in$ *Index,* $\mathbb{S}'$ *and* A', *such that the following holds:*
     (a) *either* $j < i$, $\mathbb{S}' = \mathbb{S}$ *and* $A' = A$; *or* $(A, \mathbb{S}) \dashrightarrow (A', \mathbb{S}')$;
     (b) *if* $k = 0$, *then there exists* w', *such that: (where* $\mathtt{f} = S'.Q.R(\mathtt{r}_{15})$*)*
     $(S', \mathbb{S}', A', w') \models \mathtt{q}$, $A' = \bot$, $\mathtt{pc}' = \mathtt{f}+8$, *and* $\mathtt{npc}' = \mathtt{f}+12$;
     else $\mathtt{q} \models (C_{\text{as}}, S', \mathtt{pc}', \mathtt{npc}') \preccurlyeq_j^{k-1} (A', \mathbb{S}')$.

Then we give the semantics of well-formed primitive in Def. 7.

Definition 7 (Well-defined Primitive Set Semantics)**.**

$$\begin{aligned}\Psi \models C_{\text{as}} : \Omega \ ::= \ & \forall\, \mathtt{f} \in \mathrm{dom}(\Omega),\ \iota.\ \exists\, \Upsilon,\ \overline{v},\ \mathtt{fp}, \mathtt{fq}.\\ & \mathsf{wdSpec}(\mathtt{fp}, \mathtt{fq}, \Upsilon) \wedge (\mathtt{fp}\ \iota \Rightarrow (\!|\, \Upsilon(\overline{v}) \,|\!) * \mathrm{true})\\ & \wedge (C_{\text{as}}, \mathtt{f}) \preccurlyeq^{(\mathtt{fp}\ \iota, \mathtt{fq}\ \iota)} \Upsilon(\overline{v})\\ \text{where } \Omega(\mathtt{f}) = \Upsilon, \Psi(\mathtt{f}) = (\mathtt{fp}, \mathtt{fq}).\end{aligned}$$

It says that for any high-level abstract assembly primitive in primitive set Ω, we can establish a simulation relation defined in Def. 5 between its low-level implementation in code heap C_{as} and itself. Theorem 1, whose correctness can be derived from Lemmas 4 and 7, shows the soundness of our logic, which means that the extended program logic can imply the contextual refinement between implementation C_{as} and abstract assembly primitives Ω.

Soundness Proof. We show the soundness proof of our logic. We first give Lemma 3 to show that the simulation relation for instruction sequence can compose to the simulation relation for function.

Lemma 3. *If* $\Psi \models \{(\mathtt{p}, \mathtt{q})\}\ \mathtt{pc}{:}C_{\text{as}}[\mathtt{pc}]$ *and* $\models C_{\text{as}} : \Psi$, *then* $(C_{\text{as}}, \mathtt{f}) \preccurlyeq^{(\mathtt{p}, \mathtt{q})} A$.

Proof. By the definition of $(C_{\text{as}}, \mathtt{f}) \preccurlyeq^{(\mathtt{p}, \mathtt{q})} A$ (in Def. 5), we need to prove that, for all S, $\mathbb{S}$, w and A:

$$\begin{aligned}&(S, \mathbb{S}, w, A) \models \mathtt{p} \implies\\ &\qquad \exists\, i \in \mathit{Index}.\ \mathtt{q} \models (C_{\text{as}}, S, \mathtt{f}, \mathtt{f}+4) \preccurlyeq_i^0 (A, \mathbb{S})\end{aligned}$$

The index i can be instantiated as $(w, 0, |\, C_{\text{as}}[\mathtt{pc}]\,|)$, where $|\,\mathbb{I}\,|$ means getting the length of instruction sequence $\mathbb{I}$. And $(w, k, |\,\mathbb{I}\,|) < (w', k', |\,\mathbb{I}'\,|)$ is defined as (k and k' are depths of function call):

$$w < w' \vee (w = w' \wedge (k < k' \vee (k = k' \wedge |\,\mathbb{I}\,| < |\,\mathbb{I}'\,|)))$$

The simulation relation $\mathtt{q} \models (C_{\text{as}}, S, \mathtt{f}, \mathtt{f}+4) \preccurlyeq_i^0 (A, \mathbb{S})$ is defined co-inductively in Def. 6. So, we prove it by co-induction and discuss each case of the simulation respectively. □

Lemma 4 (Logic Ensures Simulation)**.**

- $\Psi \vdash \{(\mathtt{p}, \mathtt{q})\}\ \mathtt{f} : \mathbb{I} \implies \Psi \models \{(\mathtt{p}, \mathtt{q})\}\ \mathtt{f}{:}\mathbb{I}$
- $\vdash C_{\text{as}} : \Psi \Longrightarrow \models C_{\text{as}} : \Psi$
- $\Psi \vdash C_{\text{as}} : \Omega \Longrightarrow \Psi \models C_{\text{as}} : \Omega$

Proof. We prove each conclusion respectively.

- The well-formed instruction sequence shown as $\Psi \vdash \{(\mathtt{p},\mathtt{q})\}\,\mathtt{f} : \mathbb{I}$ is defined inductively in Fig. 19. We proof it by induction over the derivation of $\Psi \vdash \{(\mathtt{p},\mathtt{q})\}\,\mathtt{f} : \mathbb{I}$.

- By the definition of $\vdash C_{\text{as}} : \Psi$ (defined in Fig. 19), we get that for all $\mathtt{f} \in \text{dom}(\Psi)$, ι:

$$\Psi(\mathtt{f}) = (\mathtt{fp}, \mathtt{fq}) \tag{14}$$

$$\Psi \vdash \{(\mathtt{fp}\,\iota, \mathtt{fq}\,\iota)\}\,\mathtt{f} : C_{\text{as}}[\mathtt{f}] \tag{15}$$

  Then we unfold the proof goal $\models C_{\text{as}} : \Psi$ by its definition (in Def. 4). And we need to prove that for all $\mathtt{f} \in \text{dom}(\Psi)$, ι :

$$\Psi(\mathtt{f}) = (\mathtt{fp}, \mathtt{fq}) \tag{g1}$$

$$\Psi \models \{(\mathtt{fp}\,\iota, \mathtt{fq}\,\iota)\}\,\mathtt{f} : C_{\text{as}}[\mathtt{f}] \tag{g2}$$

  Because we have already proved that: $\Psi \vdash \{(\mathtt{p},\mathtt{q})\}\,\mathtt{f} : \mathbb{I} \implies \Psi \models \{(\mathtt{p},\mathtt{q})\}\,\mathtt{f} : \mathbb{I}$, we finish the proof by applying this lemma.

- By the definition of $\Psi \vdash C_{\text{as}} : \Omega$ (defined in Fig. 19), the following hold:

$$\vdash C_{\text{as}} : \Psi \tag{16}$$

$$\text{for all } \mathtt{f} \in \text{dom}(\Psi), \exists \Upsilon, \text{fp}, \text{fq}. \begin{cases} \Psi(\mathtt{f}) = (\mathtt{fp}, \mathtt{fq}),\ \Omega(\mathtt{f}) = \Upsilon \\ \mathsf{wdSpec}(\mathtt{fp}, \mathtt{fp}, \Upsilon) \end{cases} \tag{17}$$

  Since we have proved that $\vdash C_{\text{as}} : \Psi \implies \models C_{\text{as}} : \Psi$, we get that the following holds from (16):

$$\models C_{\text{as}} : \Psi \tag{18}$$

  Then, by unfolding $\Psi \models C_{\text{as}} : \Omega$, we need to prove that for all $\mathtt{f} \in \text{dom}(\Omega), \iota$, there exists Υ, $\overline{v}$, $\mathtt{fp}$, $\mathtt{fq}$, such that:

  – $\mathsf{wdSpec}(\mathtt{fp}, \mathtt{fq}, \Upsilon)$, $\Omega(\mathtt{f}) = \Upsilon$, $\Psi(\mathtt{f}) = (\mathtt{fp}, \mathtt{fq})$;
  – $\mathtt{fp}\,\iota \Rightarrow (\!|\, \Upsilon(\overline{v}) \,|\!) * \text{true}$;
  – $(C_{\text{as}}, \mathtt{f}) \preccurlyeq^{(\mathtt{fp}\ \iota, \mathtt{fq}\ \iota)} \Upsilon(\overline{v})$;

  The proof of the first subgoal can be done by assumption (17) directly; the second one can be done according to the definition of $\mathsf{wdSpec}(\mathtt{fp}, \mathtt{fq}, \Upsilon)$ (defined in Def. 2), which requires that the abstract assembly primitive should be specified in the precondition; as for the third one, we first prove that the following holds by (18):

$$\Psi \models \{(\mathtt{fp}\ \iota, \mathtt{fq}\ \iota)\}\,\mathtt{f} : C_{\text{as}}[\mathtt{f}] \tag{19}$$

  Then, we finish the proof by applying Lemma 3 on (19) (18).

□

Then, we give Lemma 7, which says that our simulation for function implies primitive correctness. We introduce a whole program simulation defined in Def. 8 and divide the correctness proof of Lemma 7 into two steps. *First*, we prove that the simulation for function implies the whole program simulation in Lemma 5; *Second*, we prove that the whole program simulation implies the refinement relation between low- and high-level programs in Lemma 6.

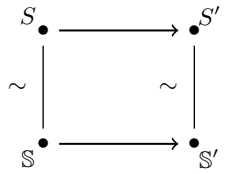
Definition 8 (Whole program simulation)**.** *Whenever* $\text{P} \leqslant^i \mathbb{P}$ *holds, the following are true:*

1. *if* $\text{P} ::\xRightarrow{\tau} \text{P}'$*, then one of the following holds:*
   - $\exists j < i.\, \text{P}' \leqslant^j \mathbb{P}$*; or*
   - $\exists j, \mathbb{P}'.\, \mathbb{P} :\xRightarrow{\tau}{}^{+} \mathbb{P}'$*, and* $\text{P}' \leqslant^j \mathbb{P}'$*;*
2. *if* $\text{P} ::\xRightarrow{e} \text{P}'$*, then* $\exists j, \mathbb{P}'.\, \mathbb{P} :\xRightarrow{e}{}^{+} \mathbb{P}'$*, and* $\text{P} \leqslant^j \mathbb{P}$*;*
3. *if* $\text{P} ::\xRightarrow{\tau} \textbf{abort}$*, then* $\mathbb{P} :\xRightarrow{\tau}{}^{+} \textbf{abort}$*.*

Lemma 5. *If* $\Psi \models C_{as} : \Omega$*,* $S \sim \mathbb{S}$*,* $\mathsf{ProgSafe}((C, \Omega), \mathbb{S})$ *and* $\mathbb{S}.\mathcal{K} = (\_, \mathtt{pc}, \mathtt{npc})$*, then there exists* $i \in \mathit{Index}$*, such that* $(C \uplus C_{\text{as}}, S, \mathtt{pc}, \mathtt{npc}) \leqslant^i ((C, \Omega), \mathbb{S})$*.*

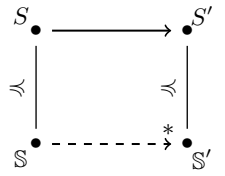
Proof. Since the whole program simulation is defined co-inductively, we prove this lemma by co-induction. We discuss whether the low-level program is executing the command c in client code C or the implementations of abstract assembly primitives C_{as}.

- If $c \in \mathrm{dom}(C)$, we use the following figure to illustrate such case:



  Since the client code of low- and high-level programs are the same, the high-level will execute the same command c. We can prove that the state relation, shown as $S \sim \mathbb{S}$ and defined in Sec. 4.3, can be preserved after the execution, shown as $S' \sim \mathbb{S}'$. So, we establish the simulation relation between low- and high-level programs.

- If $c \in \mathrm{dom}(C_{\mathrm{as}})$, we use the following figure to illustrate such case:



  The assumption $\Psi \models C_{\mathrm{as}} : \Omega$ defined in Def. 7 says that each SPARCv8 function has a simulation relation with its corresponding abstract assembly primitive. Such simulation relation (shown as $\preccurlyeq$ in the above figure) ensures that for each step of low-level execution, the high-level program will execute zero or one step. So we establish the simulation relation between low- and high-level programs.

The assumption $\mathsf{ProgSafe}((C,\Omega),\mathbb{S})$ makes sure that the client code doesn't include instruction `save`, `restore` and `wr`, because we do not give semantics to them in high-level as explained in Sec. 4.1. □

Lemma 6. $\mathrm{P} \leqslant^i \mathbb{P} \implies \mathrm{P} \subseteq \mathbb{P}$

Lemma 7 (Simulation Implies Primitive Correctness)**.**

$$\Psi \models C_{\mathrm{as}} : \Omega \Longrightarrow C_{\mathrm{as}} \sqsubseteq \Omega$$

Proof. By the definition of $C_{\mathrm{as}} \sqsubseteq \Omega$ (in Def. 1), we need to prove that for any S, $\mathbb{S}$, C, `pc` and `npc`:

$$(S \sim \mathbb{S} \wedge \mathsf{ProgSafe}((C,\Omega),\mathbb{S}) \wedge \mathbb{S}.\mathcal{K} = (\_, \mathtt{pc}, \mathtt{npc})) \\ \Longrightarrow (C \uplus C_{\mathrm{as}}, S, \mathtt{pc}, \mathtt{npc}) \subseteq ((C,\Omega),\mathbb{S})$$

Here, the C is the client code, and S and $\mathbb{S}$ represent the initial states of low- and high-level program respectively. By Lemma 5, we know there exisits $i \in \mathit{Index}$, such that:

$$(C \uplus C_{\mathrm{as}}, S, \mathtt{pc}, \mathtt{npc}) \leqslant^i ((C,\Omega),\mathbb{S}) \qquad (20)$$

By applying Lemma 6 on (20), the proof is done. □

Theorem 1 (Logic Soundness)**.**

$$\Psi \vdash C_{\mathrm{as}} : \Omega \Longrightarrow C_{\mathrm{as}} \sqsubseteq \Omega$$

Proof. The soundness proof can be done by applying Lemma 4 and 7. □

5 Verifying Context Switch Routine

We apply our program logic to verify that a context switch routine implemented in SPARCv8, which saves the current task's context and restores the new task's context, contextually refines the switch primitive defined in Sec. 4.1. Fig. 20 shows the structure of the code.

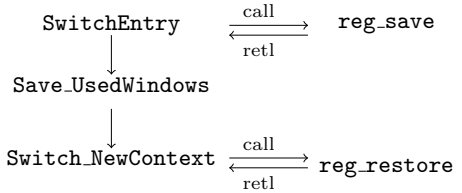


Fig.20. The Structure of Context Switch Routine

- `SwitchEntry` is the entry of the context switch routine. It saves the local and in registers of current window into stack (in memory), and calls `reg_save` to save other registers into TCB.

- `Save_UsedWindows` saves the register windows (except the current one) into the current task's stack in memory.
- `Switch_NewContext` restores the general registers from the new task's TCB (by calling `reg_restore`) and its stack in memory respectively. Then it sets the new task as the current one.

The main complexity of the verification lies in the code that manages the register window. To save all the used register windows, `Save_UsedWindows` repetitively restores the next window into general registers (as the current window) and then saves them into memory, until all the windows are saved.

Specification. Below we give the pre- and post-conditions (a_{pre} and a_{post}) of the verified module. Each of them takes 6 arguments, the id of the current task t_c, the id of the new task t_n, the values *env* of general registers and other register windows saving contexts, the new task's context *nst* to be restored, and the thread local state $\mathcal{K}_c$ of current task and $\mathcal{K}_n$ of the new task.

$$
\begin{array}{l}
a_{pre}(\mathsf{t}_c, \mathsf{t}_n, env, nst, \mathcal{K}_c, \mathcal{K}_n) \ ::= \\
\quad \mathsf{Env}(env) * (\mathsf{TaskNew} \Mapsto (\mathsf{t}_n, 0)) * \blacklozenge(10) * \\
\qquad \mathsf{CurT}(\mathsf{t}_c, \_, env, \mathcal{K}_c) * \mathsf{RdyT}(\mathsf{t}_n, nst, \mathcal{K}_n) * (\!|\, \mathsf{switch}(\mathsf{nil}) \,|\!) \\
a_{post}(\mathsf{t}_c, \mathsf{t}_n, env, nst, \mathcal{K}_c, \mathcal{K}_n) \ ::= \\
\quad \exists\, env', \mathcal{K}'.\, \mathsf{Env}(env') * (\mathsf{TaskNew} \Mapsto (\mathsf{t}_n, 0)) * \\
\qquad (\mathsf{CurT}(\mathsf{t}_n, nst, env', \mathcal{K}') \wedge \mathsf{p\_env}(env') = nst) * \\
\qquad\quad \mathsf{RdyT}(\mathsf{t}_c, \mathsf{p\_env}(env), \mathcal{K}_c) * (\!|\, \bot \,|\!)
\end{array}
$$

In the specification, we use $\mathsf{Env}(env)$ to specify the values of general registers and the register windows. We describe the state of the current task using $\mathsf{CurT}(\mathsf{t}_c, \_, env, \mathcal{K}_c)$. It describes the memory of current task's TCB and stack for saving contexts in low-level, the thread local state $\mathcal{K}_c$ in high-level, and the state relation between the current thread t_c in low- and high-level. Similarly, $\mathsf{RdyT}(\mathsf{t}_n, nst, \mathcal{K}_n)$ describes states of new task t_n in low- and high-level programs and their relation. The memory location $\mathsf{TaskNew}$ records the identifier of the new task t_n. And we use $\mathsf{TaskNew} \Mapsto (\mathsf{t}_n, 0)$ to denote that $\mathsf{TaskNew}$ saves $(\mathsf{t}_n, 0)$ in both low- and high-level memory.

$$l \Mapsto v \ ::= (l \mapsto v) * (l \rightarrowtail v)$$

The precondition takes ten tokens ($\blacklozenge(10)$). As we have explained, verifying instruction `call` and `jmp` will consume a token. So, verifying function call `reg_save` and `reg_restore` will both consume a token. And `Save_Usedwindows`, which saves the context of each previous window into memory repetitively until the invalid one, will execute at most eight times, because the upper bound of the number of windows is eight. So, ten tokens is sufficient (two for `reg_save` and `reg_restore`, and eight for `Save_Usedwindows`).

If we compare a_{pre} and a_{post}, we can see that t_n becomes the current task ($\mathsf{CurT}(\mathsf{t}_n, nst, env', \mathcal{K}')$), and its general registers and stack, specified by $\mathsf{Env}(env')$, are loaded from the saved context *nst* (*i.e.* $\mathsf{p\_env}(env') = nst$). Here $\mathsf{p\_env}(env')$ refers to the part of the environment that we want to save or restore as context. Correspondingly, t_c becomes a non-current-thread, and part of its environment *env* at the entry of the context switch is saved, as specified by $\mathsf{RdyT}(\mathsf{t}_c, \mathsf{p\_env}(env), \mathcal{K}_c)$. The execution of switch should be done in the final state. We use $\mathcal{K}'$ to represent the thread local state of t_n instead of $\mathcal{K}_n$ in the final state, because the execution of switch will modify the program counters in $\mathcal{K}_n$.

Proof outline We show how to use our relational program logic defined in Fig. 19 to verify the correctness of the context switch routine. We first instantiate the set of abstract assembly primitives (21) and the code heap specification (22) below:

$$\Omega \ ::= \{\texttt{SwitchEntry} \rightsquigarrow \mathsf{switch}\} \tag{21}$$

$$\begin{array}{l}\Psi ::= \{\texttt{SwitchEntry} \rightsquigarrow (a_{pre}, a_{post}),\\ \quad\texttt{reg\_save} \rightsquigarrow (\mathbb{fp}_{rs}, \mathbb{fq}_{rs}),\\ \quad\texttt{reg\_restore} \rightsquigarrow (\mathbb{fp}_{rr}, \mathbb{fq}_{rr})\\ \quad\texttt{Save\_Usedwindows} \rightsquigarrow (\mathbb{fp}_{su}, \mathbb{fq}_{su}),\\ \quad\texttt{Switch\_NewContext} \rightsquigarrow (\mathbb{fp}_{sn}, \mathbb{fq}_{sn})\}\end{array} \quad (22)$$

The set of abstract assembly primitive Ω contains only one abstract assembly primitive switch. And the code heap specification Ψ contains the specifications of each code block. We use $(\mathbb{fp}_{rs}, \mathbb{fq}_{rs})$, $(\mathbb{fp}_{rr}, \mathbb{fq}_{rr})$, $(\mathbb{fp}_{su}, \mathbb{fq}_{su})$ and $(\mathbb{fp}_{sn}, \mathbb{fq}_{sn})$ to represent the specifications of `reg_save`, `reg_restore`, `Save_Usedwindows` and `Switch_NewContext` respectively. Since the post-condition in our logic specifies the state when the *current function returns*, the specification of `SwitchEntry` is (a_{pre}, a_{post}).

First, we prove that the specification of context switch routine is well-defined in Lemma 8.

Lemma 8. $\mathsf{wdSpec}(a_{pre}, a_{post}, \mathsf{switch})$

Proof. By the definition of wdSpec (defined in Def. 2), we need to prove the following:

- for any $\overline{v}, \mathbb{S}, \mathbb{S}', \mathbb{S}_r$. if $\mathsf{switch}(\overline{v})(\mathbb{S})(\mathbb{S}')$, and $\mathbb{S} \perp \mathbb{S}_r$, then the following holds :
  - $\mathbb{S}'.\mathcal{K}.\texttt{pc} = \texttt{f} + 8$, $\mathbb{S}'.\mathcal{K}.\texttt{npc} = \texttt{f} + 12$ (where $\mathbb{S}'.\mathcal{K}.\mathbb{Q}.\mathbb{R}(\texttt{r}_{15}) = \texttt{f}$);
  - there exists $\mathbb{S}'', \mathbb{S}'_r$, $\mathsf{switch}(\overline{v})(\mathbb{S} \uplus \mathbb{S}_r)(\mathbb{S}'')$, $\mathbb{S}'' = \mathbb{S}' \uplus \mathbb{S}'_r$, and $\mathbb{S}_r.T = \mathbb{S}'_r.T$, $\mathbb{S}_r.M = \mathbb{S}'_r.M$;

  Prove by the definition of switch, which ensures that, in the final state, the program counters will point to the correct pointers and the threads and memory in $\mathbb{S}_r$ remain unchanged, because the execution of switch only accesses the current and new tasks in thread pool and the location TaskNew in memory.

- for any $\texttt{t}_c, \texttt{t}_n, env, nst, \mathcal{K}_c, \mathcal{K}_n$,
  - $a_{pre}(\texttt{t}_c, \texttt{t}_n, env, nst, \mathcal{K}_c, \mathcal{K}_n) \Rightarrow (\!|\mathsf{switch}|\!) * \mathrm{true}$;
  - $a_{post}(\texttt{t}_c, \texttt{t}_n, env, nst, \mathcal{K}_c, \mathcal{K}_n) \Rightarrow (\!|\bot|\!) * \mathrm{true}$;

  Prove by the definition of a_{pre} and a_{post}.

- for any $\overline{v}, S, \mathbb{S}$, if $(S, \mathbb{S}, \_, \_) \in \mathsf{INV}(\mathsf{switch}(\overline{v}), \overline{v})$, then there exists $\texttt{t}_c, \texttt{t}_n, env, nst, \mathcal{K}_c, \mathcal{K}_n, \mathbb{p}_r$ and w, such that:
  - $(S, \mathbb{S}, \mathsf{switch}(\overline{v}), w) \models a_{pre}(\texttt{t}_c, \texttt{t}_n, env, nst, \mathcal{K}_c, \mathcal{K}_n) * \mathbb{p}_r$;
  - $(a_{post}(\texttt{t}_c, \texttt{t}_n, env, nst, \mathcal{K}_c, \mathcal{K}_n) * \mathbb{p}_r) \Rightarrow \mathsf{INV}(\bot, \_)$;
  - $\mathsf{Sta}(\mathsf{switch}(\overline{v}), \mathbb{p}_r)$.

  Prove by the definitions of a_{pre} and a_{post}. They specify the states of task $\texttt{t}_c$ and $\texttt{t}_n$ and memory location TaskNew in low- and high-level. And we define the frame $\mathbb{p}_r$ to depict the state of the remaining tasks and memory (excluding tasks $\texttt{t}_c$, $\texttt{t}_n$, and memory location TaskNew) in low- and high-level. Then, we can prove that the state relation between low- and high-level holds at the entry and exit of the context switch routine. Since the execution of switch only accesses the current and new tasks in thread pool, and the memory location TaskNew, the threads and memory specified in assertion $\mathbb{p}_r$ remain unchanged in the final state and $\mathbb{p}_r$ keeps stable ($\mathsf{Sta}(\mathsf{switch}, \mathbb{p}_r)$).

□

We use $C_{\texttt{switch}}$ to represent the code heap storing the code of context switch routine, which includes the code blocks `SwitchEntry`, `reg_save`, `reg_restore`, `Save_Usedwindows` and `Switch_NewContext`. We prove that the $C_{\texttt{switch}}$ is well-defined in Lemma 9.

Lemma 9. $\vdash C_{\texttt{switch}} : \Psi$

Proof. The code heap specification Ψ have been defined in (22). We unfold $\vdash C_{\texttt{switch}} : \Psi$ according to its definition (in Fig. 19). And we need to prove that for any ι_1, ι_2, ι_3, ι_4 and ι_5, the following hold (we use $\texttt{f}_{se}$, $\texttt{f}_{rs}$, $\texttt{f}_{rr}$, $\texttt{f}_{su}$ and $\texttt{f}_{sn}$ to represent the starting labels of `SwitchEntry`, `reg_save`, `reg_restore`, `Save_Usedwindows` and `Switch_NewContext` below):

$$\Psi \vdash \{(a_{pre}\ \iota_1, a_{post}\ \iota_1)\}\ \texttt{f}_{se} : C_{\texttt{switch}}[\texttt{f}_{se}] \quad \text{(g-wfse)}$$

$$\Psi \vdash \{(\mathtt{fp}_{rs}\ \iota_2, \mathtt{fq}_{rs}\ \iota_2)\}\ \mathtt{f}_{rs} : C_{\mathtt{switch}}[\mathtt{f}_{rs}] \quad \text{(g-wfrs)}$$

$$\Psi \vdash \{(\mathtt{fp}_{rr}\ \iota_3, \mathtt{fq}_{rr}\ \iota_3)\}\ \mathtt{f}_{rr} : C_{\mathtt{switch}}[\mathtt{f}_{rr}] \quad \text{(g-wfrr)}$$

$$\Psi \vdash \{(\mathtt{fp}_{su}\ \iota_4, \mathtt{fq}_{su}\ \iota_4)\}\ \mathtt{f}_{su} : C_{\mathtt{switch}}[\mathtt{f}_{su}] \quad \text{(g-wfsu)}$$

$$\Psi \vdash \{(\mathtt{fp}_{sn}\ \iota_5, \mathtt{fq}_{sn}\ \iota_5)\}\ \mathtt{f}_{sn} : C_{\mathtt{switch}}[\mathtt{f}_{sn}] \quad \text{(g-wfsn)}$$

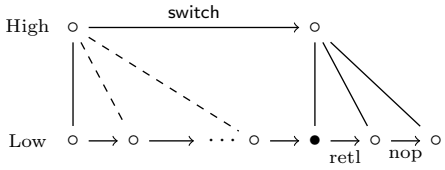


Fig.21. Point doing refinement reasoning

The correctness proof from (g-wfse) to (g-wfsn) can be done by applying the inference rules for code block shown in Fig. 19. We can choose any place to apply **ABSCSQ** rule to execute primitive switch. Here, we apply **ABSCSQ** rule in verifying the code block `Switch_NewContext` (g-wfsn), when context switch routine returns, as shown in Fig. 21. In Fig. 21, we use the solid circle to represent the point applying **ABSCSQ** rule, and after the execution of switch, the state relation between low- and high-level can be reestablished and we use solid line to represent that such relation holds in Fig. 21. □

Theorem 2 (Context Switch Routine Correctness)**.** $\Psi \vdash C_{\mathtt{switch}} : \Omega$.

Proof. By the definition of $\Psi \vdash C_{\mathtt{switch}} : \Omega$, where Ω and Ψ are defined in (21) and (22) respectively, we prove the following:

- $\vdash C_{\mathrm{as}} : \Psi$. Prove by applying Lemma 9.
- $\mathsf{wdSpec}(a_{pre}, \mathsf{a}_{post}, \mathsf{switch})$. Prove by applying Lemma 8.

□

This part of work has not been mechanized in Coq. In our conference paper, we show that we apply our logic that do not support refinement verification to verify the main body of the context switch routine in a realistic embedded OS kernel for aerospace crafts, which consists of around 250 lines of SPARCv8 code, by 6690 lines of Coq proof scripts. Here, the context switch routine verified by applying our relational program logic is a simplified version of context switch routine, which omits some details like judging whether the current thread is a valid thread. Verifying that each code block is well-defined by applying the inference rules in our new logic is no different from the previous proof work. The additional proof effort includes: (1) proving that the specification of context switch routine is well-defined (presented in Lemma 8); (2) applying **ABSCSQ** rule to execute primitive switch and proving that the state relation between low- and high-level programs can be reestablished in verifying code block `Switch_NewContext`, when context switch routine returns (as noted in the proof of Lemma 9).

We present more details of proof in TR [8].

6 Related Work and Conclusion

There has been much work on assembly or machine code verification. Most of them do not support function calls or simply treat function calls in the continuation-passing style where return addresses are viewed as first class code pointers [13, 14, 15, 16, 17, 18, 19]. SCAP [7] supports assembly code verification with various stack-based control abstractions, including function call and return. We follow the same idea here. However, SCAP gives a syntactic-based soundness proof by establishing the preservation of the syntactic judgment, which makes it difficult to interact with other modules verified in different logic. Since our goal is to verify inline assembly and link the verified code with the verified C programs, we give a direct-style semantic model of the logic judgments. And it allows us to extend our program logic to support verifying contextual refinement without much challenges. Also SCAP is based on a sim-

plified subset of assembly instructions, while our work is focused on a realistically modeled subset of SPARCv8 instructions.

In terms of the support of realistic instruction sets, previous work on proof-carrying code (PCC) and typed assembly language (TAL) mostly supports subsets of x86. Myreen's work [20] presents a framework for ARM verification based on a realistic model (but it doesn't support function call and return).

As part of the Foundational Proof-Carrying Code (FPCC) project [14], Tan and Appel present a program logic $\mathcal{L}_c$ for reasoning about control flow in assembly code [19]. Although $\mathcal{L}_c$ is implemented on top of the SPARC machine language, the underlying logic is a type system instead of a full-blown program logic for functional correctness. It reasons about functions in the continuation-passing style. Also handling SPARC features such as delayed writes or delayed control transfers is not the focus of $\mathcal{L}_c$. There has been work on mechanized semantics of the SPARCv8 ISA. Hou *et al.* [21] model the SPARCv8 ISA in Isabelle/HOL, and test their formal model against LENON3 simulation board, which is a synthesisable VHDL model of a 32-bit processor compliant with the SPARCv8 architecture, through more than 100,000 instruction instances. Wang *et al.* [10] formalize its semantics in Coq. Our operational semantics of SPARCv8 follows Wang *et al.* [10]. Since neither Wang *et al.* nor we validate the formalization against actual hardware, this remains as future work.

Ni *et al.* [22] verify a context switch module of 19 lines in x86 code to showcase the support of embedded code pointers (ECP) in XCAP [18]. We use our program logic to verify the contextual refinement between a context switch routine in SPARCv8 and switch primitive. The context switch routine implemented in SPARCv8 that we verified is more complicated then implemented in x86, because of the requirement to save the contexts stored in register window in memory.

Yang and Hawblitzel [23] verify Verve, an x86 implementation of an experimental operating system. Verve has two levels, the high-level TAL code and the low-level "Nucleus" that provides primitive access to hardware and memory. The Nucleus code is verified automatically using the Z3 SMT solver, while the goal of our work is to generate machine checkable proofs. Another key difference is the use of different ISAs. Here we give details to verify specific features of SPARCv8 programs.

There have been many techniques and tools proposed for automated program verification (*e.g.* [24, 25]). It is possible to adapt them to verify SPARCv8 code. We propose a new program logic and do the verification in Coq mainly because the work is part of a big project for a fully certified OS kernel for aerospace crafts whose inline assembly is written in SPARCv8. We already have a program logic implemented in Coq for C programs, which allows us to verify C code with Coq proofs. Therefore we want to have a program logic for SPARCv8 so that it can be linked with the correctness proof of C and can generate machine-checkable Coq proofs too. That said, many of the automated verification techniques can be applied to reduce the manual efforts to write Coq proofs, which we would like to study in the future work.

Conclusion and future work. We present a relational program logic for SPARCv8. Our logic is based on a realistic semantics model and supports the main features of SPARCv8, including delayed control transfer, delayed writes, and register windows. It also supports modular reasoning of function calls in a direct-style and refinement verification. We apply our logic to verify that there is a contextual refinement between the

context switch routine implemented in SPARCv8 and the switch primitive for task switching.

Our current work can only handle sequential SPARCv8 program verification and do not consider interrupts in the machine model. We would like to extend it for concurrency verification and finish the step ① shown in Fig. 2 in the future. The step ① says that the compilation ensures that the behaviors of the Pseudo-SPARCv8 code calling abstraction assembly primitives in intermediate level refine the behaviors of the client C code calling abstract assembly primitives in the source level. We also consider automating the reasoning for certain parts with Coq tactics [26] or automatic provers like Z3 [27] to reduce the required human effort in the verification work in the future.

References

[1] Linux v2.6.17.10. `https://elixir.bootlin.com/linux/v2.6.17.10/source/arch`.

[2] Fengwei Xu, Ming Fu, Xinyu Feng, Xiaoran Zhang, Hui Zhang, and Zhaohui Li. A practical verification framework for preemptive os kernels. In *CAV*, pages 59–79, July 2016.

[3] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS Kernel. In *SOSP*, pages 207–220, Oct 2009.

[4] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. Deep specifications and certified abstraction layers. In *POPL*, pages 595–608, Jan 2015.

[5] SPARC. `https://gaisler.com/doc/sparcv8.pdf`.

[6] Junpeng Zha, Xinyu Feng, and Lei Qiao. Modular Verification of SPARCv8 Code. In *APLAS*, pages 245–263, December 2018.

[7] Xinyu Feng, Zhong Shao, Alexander Vaynberg, Sen Xiang, and Zhaozhong Ni. Modular Verification of Assembly Code with Stack-Based Control Abstractions. In *PLDI*, June 2006.

[8] Technical report and Coq implementations. `https://github.com/jpzha/VeriSparc`.

[9] Xavier Leroy and Sandrine Blazy. Formal verification of a C-like memory model and its uses for verifying program transformations. *Journal of Automated Reasoning*, 41(1):1–31, July 2008.

[10] Jiawei Wang, Ming Fu, Lei Qiao, and Xinyu Feng. Formalizing SPARCv8 Instruction Set Architecture in Coq. In *SETTA*, Oct 2017.

[11] J.C. Reynolds. Separation logic: A logic for shared mutable data structures. July 2002.

[12] Hongjin Liang, Xinyu Feng, and Zhong Shao. Compositional verification of termination-preserving refinement of concurrent programs. In *LICS*, July 2014.

[13] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Proc.2nd USENIX Symp. on Operating System Design and Impl*, pages 229–243, 1996.

[14] Andrew W. Appel. Foundational proof-carrying code. In *Proc. 16th Annual IEEE Symposium on Logic in Computer Science*, pages 85–97, Jan 1998.

[15] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. Talx86: A realistic typed assembly language. In *1999 ACM SIGPLAN Workshop on Compiler Support for System Software*, pages 25–35, May 1996.

[16] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. In *POPL*, pages 85–97, Jan 1998.

[17] Dachuan Yu, A. Hamid Nadeem, and Zhong Shao. Building certified libraries for PCC : Dynamic storage allocation. *Science of Computer Programming*, 50(1-3):101–127, Mar 2004.

[18] Zhaozhong Ni and Zhong Shao. Certified assembly programming with embedded code pointers. In *POPL*, pages 320–333, 2006.

[19] Gang Tan and Andrew W. Appel. A compositional logic for control flow. In *VMCAI*, Jan 2006.

[20] Magnus O. Myreen and Michael J.C. Gordon. Hoare logic for realistically modelled machine code. In *Proc. 13th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, 2007.

[21] Hou Zhe, David Sanan, Alwen Tiu, Yang Liu, and Koh Chuen Hoa. An Executable Formalisation of the SPARCv8 Instruction Set Architecture: A Case Study for the LEON3 Processor. In *FM*, 2016.

[22] Zhaozhong Ni, Dachuan Yu, and Zhong Shao. Using XCAP to Certify Realistic Systems code: Machine context management. In *TPHOLs*, Sept 2007.

[23] Jean Yang and Chris Hawblitzel. Safe to the last instruction: Automated verification of a type-safe operating system. In *PLDI*, pages 99–110, 2010.

[24] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic execution with separation logic. In *APLAS*, 2005.

[25] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, 2005.

[26] Jingyuan Cao, Ming Fu, and Xinyu Feng. Practical tactics for verifying c programs in coq. In *CPP*, pages 97–108, January 2015.

[27] Z3. `https://github.com/Z3Prover/z3`.

A More about High-level Instructions Execution

We give some supplements about the execution of high-level instructions. As we have explained in Sec. 4.1, the register windows and delayed buffer in pyhsical SPARCv8 program state are omitted in high-level Pseudo-SPARCv8 program state. So, we do not define state transtion rules for instructions `save`, `restore`, `rd`, and `wr`. The instruction transition rules for the rest of instructions, like `ld` and `add`, have no much difference with the rules in pyhical SPARCv8 program.

$$\frac{[\![\mathtt{a}]\!]_{\mathbb{R}} = l \qquad M(l) = v \qquad \mathbb{R}' = \mathbb{R}\{\!|\, \mathtt{r}_d \rightsquigarrow v \,|\!\}}{\mathbf{exec}(\mathtt{ld\ a\ r}_d, ((\mathbb{R},\mathbb{F}),M), ((\mathbb{R}',\mathbb{F}),M))}$$

$$\frac{\mathbb{R}(\mathtt{r}_s) = v_1 \qquad [\![\mathtt{o}]\!]_{\mathbb{R}} = v_2 \qquad \mathtt{r}_d = \mathrm{dom}(\mathbb{R}) \qquad \mathbb{R}' = \mathbb{R}\{\!|\, \mathtt{r}_d \rightsquigarrow v \,|\!\}}{\mathbf{exec}((\mathtt{add\ r}_s, \mathtt{o}, \mathtt{r}_d, ((\mathbb{R},\mathbb{F}),M)), ((\mathbb{R}',\mathbb{F}),M))}$$

Fig.A1. Transition rules for instructions `ld` and `add` in high-level

We show the state transition rules for instructions `ld` and `add` in high-level in Fig. A1. The register file updating operation is defined formally below :

$$\mathbb{R}\{\!|\, \hat{\mathtt{rn}} \rightsquigarrow v \,|\!\} \ ::= \mathbb{R}\{\hat{\mathtt{rn}} \rightsquigarrow v\} \qquad \text{where } \hat{\mathtt{rn}} \notin \{\%\mathtt{sp}, \%\mathtt{fp}\}$$

According to the definition, we can find that updating the register %`sp` (alias of $\mathtt{r}_{14}$), which is used to point to the top of the current stack frame, and %`fp` (alias of $\mathtt{r}_{14}$), which is used to point to the top of the previous stack frame is not allowed. Only the execution of instructions `Psave`, which is used to allocate a new stack frame, and `Prestore`, which is used to free the current stack frame, can modify them. The evaluation of the opand and address expression in high-level is defined formally below :

$$[\![\mathtt{o}]\!]_{\mathbb{R}} ::= \begin{cases} R(r) & \textit{if } \mathtt{o} = r \\ w & \textit{if } \mathtt{o} = w, \\ & \quad -4096 \le w \le 4095 \\ \bot & \text{otherwise} \end{cases} \qquad [\![\mathtt{a}]\!]_{\mathbb{R}} ::= \begin{cases} [\![\mathtt{o}]\!]_{\mathbb{R}} & \textit{if } \mathtt{a} = \mathtt{o} \\ v_1 + v_2 & \textit{if } \mathtt{a} = \mathtt{r} + \mathtt{o},\ \mathbb{R}(\mathtt{r}) = v_1 \\ & \quad \textbf{and } [\![\mathtt{o}]\!]_{\mathbb{R}} = v_2 \\ \bot & \text{otherwise} \end{cases}$$

The " `Psave` w " can be viewed as a macro of " `save` $\%\mathtt{sp}, -w, \%\mathtt{sp}$ ", and " `Prestore` " can be viewed as a marco of " `restore` $\%\mathtt{g}_0, \%\mathtt{g}_0, \%\mathtt{g}_0$ ".

Fig. A2 gives a simple comparision with the realistic SPARCv8 code and our Pseudo SPARCv8 code in high-level. Fig. A2 (a) is the realistic SPARCv8 code. It uses instruction " `save` $\%\mathtt{sp}, -128, \%\mathtt{sp}$ " to store the caller's context and allocate a new stack frame size 128 bytes for the current procedure, and use instruction "

`save`	$\%\mathtt{sp}, -128, \%\mathtt{sp}$		`Psave`	128	
`add`	$\%\mathtt{i}_0, \%\mathtt{i}_1, \%\mathtt{i}_0$		`add`	$\%\mathtt{i}_0, \%\mathtt{i}_1, \%\mathtt{i}_0$	
`ret`			`ret`		
`restore`	$\%\mathtt{g}_0, \%\mathtt{g}_0, \%\mathtt{g}_0$		`Prestore`		
	(a)			(b)	

Fig.A2. Realistic SPARCv8 Code and Pseudo-SPARCv8 Code

`restore` $\%\mathtt{g}_0, \%\mathtt{g}_0, \%\mathtt{g}_0$ " to restore the caller's context at the exitance of the current procedure. Fig. A2 (b) is the same function in Pseudo-SPARCv8 code, and we can find that the instructions that is responsible for saving and restoring the context of caller is replaced by " `Psave` 128 " and " `Prestore` ".

B More about Low-level Language

The machine states and syntax low-level SPARCv8 language (defined in Fig. A3) are taken from the model of SPARCv8 defined in Fig. 3. So, we omit some definitions, like RegName and DelayCycle, which are same as ones defined in Fig. 3 here.

$$\begin{array}{rrcl@{\qquad}rrcl}
\text{(LProg)} & \mathrm{P} & ::= & (C, S, \mathtt{pc}, \mathtt{npc}) & \text{(LState)} & S & ::= & (M, Q, D) \\
\text{(LRstate)} & Q & ::= & (R, F) & \text{(LRegFile)} & R & ::= & \text{RegName} \rightharpoonup \text{Val} \\
\text{(LFrmList)} & F & ::= & \text{nil} \mid \text{fm} :: F & \text{(LFrame)} & \text{fm} & ::= & [v_0, \ldots, v_7] \\
\text{(DBuf)} & D & ::= & \text{nil} \mid (t, X, v) & \text{(LMsg)} & \beta & ::= & \tau \mid \mathsf{out}(v)
\end{array}$$

Fig.A3. Machine States and Syntax for Low-level SPARCv8 Language

The low-level program P is a tuple including the code heap C, low-level program state S, program counter `pc` and `npc`. The code heap C is defined in Fig. 14. The low-level program state S uses the block-based memory model M, which is the same as the high-level program. The low-level message does not need $\mathsf{call}(\mathtt{f}, \overline{v})$, because the low-level program does not call abstract assembly primitive, but call its corresponding implementation, which is a function.

Operational Semantics for Low-level Code. The operational semantics for low-level program is defined in Fig. A4. Most of the state transition rules are taken from Fig. 7. Here, we use "$\mathbf{exec}_{\mathsf{L}}(\mathtt{i}, \_, \_)$" to represent the step for simple instruction `i`.

The execution of instruction `Psave` is discussed in divided into two cases : (1) if we can successfully set the next register window as the current one (represented as $\mathbf{save}(R, F) = (R', F')$), a new stack frame in memory will be allocated (shown as $\boldsymbol{alloc}(M, b, 0, w) = M'$); (2) if we can't set the next register window as the current one (represented as $\mathbf{save}(R, F) = \mathbf{undefined}$), a windows overflow trap will be triggered, and we redo the instruction `Psave`. The execution of instruction `Prestore` does the reverse. Here, we use "$\_ \uparrow\uparrow \_$" to represent the state transition caused by window overflow trap and use "$\_ \downarrow\downarrow \_$" to represent the state transition caused by window underflow trap. Their formal definitions are shown in Fig. A6.

$$\frac{(R,F) \rightrightarrows (R',F') \qquad C \vdash ((M,(R',F),D'),\mathtt{pc},\mathtt{npc}) \circ\!\xrightarrow{\beta}\! (M',(R'',F'),D'')}{(C,(M,(R,F),D),\mathtt{pc},\mathtt{npc}) ::\!\xRightarrow{\beta}\! (C,(M,(R'',F'),D''))}$$

(a) Low-level Program Transition

$$\frac{C(\mathtt{pc}) = \mathtt{i} \qquad \mathbf{exec}_{\mathsf{L}}(\mathtt{i},(M,Q,D)),(M',Q',D'))}{C \vdash ((M,Q,D),\mathtt{pc},\mathtt{npc}) \circ\!\xrightarrow{\tau}\! ((M',Q',D'),\mathtt{npc},\mathtt{npc}+4)}$$

$$\frac{C(\mathtt{pc}) = \mathtt{jmp\ a} \qquad [\![\mathtt{a}]\!]_R = \mathtt{f}}{C \vdash ((M,(R,F),D),\mathtt{pc},\mathtt{npc}) \circ\!\xrightarrow{\tau}\! ((M,(R,F),D),\mathtt{npc},\mathtt{f})}$$

$$\frac{C(\mathtt{pc}) = \mathtt{call\ f} \qquad \mathtt{r}_{15} \in \mathrm{dom}(R)}{C \vdash ((M,(R,F),D),\mathtt{pc},\mathtt{npc}) \circ\!\xrightarrow{\tau}\! ((M,(R\{\mathtt{r}_{15} \rightsquigarrow \mathtt{pc}\},F),D),\mathtt{npc},\mathtt{f})}$$

$$\frac{C(\mathtt{pc}) = \mathtt{retl} \qquad R(\mathtt{r}_{15}) = \mathtt{f}}{C \vdash ((M,(R,F),D),\mathtt{pc},\mathtt{npc}) \circ\!\xrightarrow{\tau}\! ((M,(R,F),D),\mathtt{npc},\mathtt{f}+8)}$$

$$\frac{C(\mathtt{pc}) = \mathtt{print} \qquad R(\%\mathtt{o}_0) = v}{C \vdash ((M,(R,F),D),\mathtt{pc},\mathtt{npc}) \circ\!\xrightarrow{\mathsf{out}(v)}\! ((M,(R,F),D),\mathtt{npc},\mathtt{npc}+4)}$$

$$\frac{C(\mathtt{pc}) = \mathtt{Psave}\ w \qquad \mathbf{save}(R,F) = \mathbf{undefined} \qquad (M,R,F) \Uparrow (M',R',F')}{C \vdash ((M,(R,F),D),\mathtt{pc},\mathtt{npc}) \circ\!\xrightarrow{\tau}\! ((M',(R',F'),D),\mathtt{pc},\mathtt{npc})}$$

$$\frac{C(\mathtt{pc}) = \mathtt{Prestore} \qquad \mathbf{restore}(R,F) = \mathbf{undefined} \qquad (M,R,F) \Downarrow (M',R',F')}{C \vdash ((M,(R,F),D),\mathtt{pc},\mathtt{npc}) \circ\!\xrightarrow{\tau}\! ((M',(R',F'),D),\mathtt{pc},\mathtt{npc})}$$

(b) Low-level Control Transfer Instruction Transition

$$\frac{\boldsymbol{alloc}(M,b,0,w) = M' \qquad \mathbf{save}(R,F) = (R',F') \qquad R'' = R'\{\%\mathtt{sp} \rightsquigarrow (b,0)\}}{\mathbf{exec}_{\mathsf{L}}(\mathtt{Psave}\ w,(M,(R,F),D)),(M',(R',F'),D))}$$

$$\frac{\boldsymbol{free}(b,M) = M' \qquad \mathbf{restore}(R,F) = (R',F')}{\mathbf{exec}_{\mathsf{L}}(\mathtt{Prestore},(M,(R,F),D)),(M',(R'',F''),D))}$$

$$\frac{\mathbf{save}(R,F) = (R'',F') \qquad [\![\mathtt{o}]\!]_R = v \qquad R'' = R'\{\mathtt{r}_d \rightsquigarrow R(\mathtt{r}_s)+v\}}{\mathbf{exec}_{\mathsf{L}}((\mathtt{save\ r}_s\ \mathtt{o\ r}_d,(M,(R,F),D)),(M',(R',F'),D))}$$

$$\frac{\mathbf{restore}(R,F) = (R',F') \qquad [\![\mathtt{o}]\!]_R = v \qquad R'' = R'\{\mathtt{r}_d \rightsquigarrow R(\mathtt{r}_s)+v\}}{\mathbf{exec}_{\mathsf{L}}((\mathtt{restore\ r}_s\ \mathtt{o\ r}_d,(M,(R,F),D)),(M,(R'',F'),D'))}$$

(c) Low-level Instruction Transition

$$[\![\mathtt{o}]\!]_R ::= \begin{cases} R(r) & \textit{if } \mathtt{o} = r \\ w & \textit{if } \mathtt{o} = (b,w') \textbf{ or } \mathtt{o} = w, \\ & \quad -4096 \le w \le 4095 \\ \bot & \text{otherwise} \end{cases} \qquad [\![\mathtt{a}]\!]_R ::= \begin{cases} [\![\mathtt{o}]\!]_R & \textit{if } \mathtt{a} = \mathtt{o} \\ v_1+v_2 & \textit{if } \mathtt{a} = \mathtt{r}+\mathtt{o},\ R(\mathtt{r})=v_1 \\ & \quad \textbf{and } [\![\mathtt{o}]\!]_R = v_2 \\ \bot & \text{otherwise} \end{cases}$$

(d) Low-level Expression Semantics

Fig.A4. Selected operational semantics rules for low-level program

$$\mathbf{fresh}(b, M) \ ::= \forall\, w.\, (b, w) \notin \mathrm{dom}(M)$$

$$\begin{aligned}\boldsymbol{alloc}(M, b, w_l, w_h) = M' \ ::= \ & (M' = M \ \wedge \ w_l = w_h) \vee \\ & (M' = M\{(b, w_l) \rightsquigarrow \_, \ldots, (b, w_h - 1) \rightsquigarrow \_\} \ \wedge \ \mathbf{fresh}(b, M) \ \wedge \ w_l < w_h)\end{aligned}$$

$$\boldsymbol{free}(b, M) = M' \ ::= \forall\, b' \neq b, w'.\ M'(b', w') = M(b', w') \ \wedge \ \nexists\, w.\ (b, w) \in \mathrm{dom}(M)$$

Fig.A5. Auxiliary Definitions for Memory Operation

$$\frac{\begin{array}{c} F = F_1 \cdot \mathrm{fm}_1 \cdot \mathrm{fm}_2 \cdot \mathrm{fm}_3 \cdot \mathrm{fm}_4 \qquad \mathrm{fm}_1[6] = (b, 0) \\ R(\mathtt{wim}) = 2^n \qquad \{(b, 0), \ldots, (b, 15)\} \subseteq \mathrm{dom}(M) \qquad R' = R''\{\mathtt{wim} \rightsquigarrow 2^{\mathbf{next\_cwp}(n)}\} \\ M' = M\{[(b, 0), \ldots, (b, 7)] \rightsquigarrow \mathrm{fm}_2\}\{[(b, 8), \ldots, (b, 15)] \rightsquigarrow \mathrm{fm}_3\} \end{array}}{(M, (R, F)) \Uparrow (M', (R', F))}$$

$$\frac{\begin{array}{c} F = \mathrm{fm}_1 :: \mathrm{fm}_2 :: F'' \qquad R(\mathtt{r}_{30}) = (b, 0) \qquad R(\mathtt{wim}) = 2^n \\ \{[(b, 0), \ldots, (b, 7)] \rightsquigarrow \mathrm{fm}_1', [(b, 8), \ldots, (b, 15)] \rightsquigarrow \mathrm{fm}_2'\} \subseteq M' \\ R' = R''\{\mathtt{wim} \rightsquigarrow 2^{\mathbf{prev\_cwp}(n)}\} \qquad F' = \mathrm{fm}_1' :: \mathrm{fm}_2' :: F'' \end{array}}{(M, (R, F)) \Downarrow (M, (R', F'))}$$

Fig.A6. Windows Over- and UnderFlow

C More about State Relation Between Low- and High-level Program

After introducing the definitions of low- and high-level program, we establish the state relation between low- and high-level program in this section. Establishing their state relation is not a trivial task, because there are two major differences low- and high-level program states. **First**, all the procedures' contexts of a specific thread are saved in high-level frame list $\mathbb{F}$. However, for low-level program, part of the contexts are saved in register windows (modeled as low-level frame list F), the other part of the contexts are saved in corresponding stack frame in memory, because the number of register windows is limited; **Second**, the high-level concurrent Pseudo-SPARCv8 program is multithreaded, but the low-level SPARCv8 program does not have the concept of thread pool.

$$\frac{}{(b, \mathrm{nil}, \emptyset) \Downarrow \mathrm{nil}} \qquad \frac{M_K = \{(b, \mathrm{fm}_1, \mathrm{fm}_2)\} \uplus M_K' \qquad \mathrm{fm}_2[6] = (b', 0) \qquad (b', \mathrm{nil}, M_K') \Downarrow \mathbb{F}}{(b, \mathrm{nil}, M_K) \Downarrow (\mathrm{fm}_1, \mathrm{fm}_2) :: \mathbb{F}}$$

$$\frac{M_K = \{(b, \_, \_)\} \uplus M_K' \qquad \mathrm{fm}_2[6] = (b', 0) \qquad (b', F, M_K') \Downarrow \mathbb{F}}{(b, \mathrm{fm}_1 :: \mathrm{fm}_2 :: F, M_K) \Downarrow (\mathrm{fm}_1, \mathrm{fm}_2) :: \mathbb{F}}$$

Fig.A7. Relation for low- and high-level FrameList

Relation for low- and high-level FrameList. The relation between low- and high-level frame list is defined in Fig. A7. We represent this relation as form "$(b, F, M_K) \Downarrow \mathbb{F}$", The tuple of b, F and M_K is the state of stack in low-level program, because, in the low-level program, part of the produces' contexts are saved in frame list F, which can also be understand as a prefix the whole frame list describe in assertion $\mathtt{cwp} \mapsto (\!|\_, F|\!)$, the other part of the contexts are saved in corresponding frame list represent as M_K. The high-level frame list $\mathbb{F}$ represents the state of stack in high-level program. Fig. 12 gives a more intuition understanding of this relation. Here, some part

$$\mathsf{ctxfm}(R,F) ::= \begin{cases} F_1 & \textit{if } R(\mathtt{cwp}) = w_{id},\ R(\mathtt{wim}) = 2^n,\ \mathtt{cwp} \neq n, \\ & \quad F = F_1 \cdot F_2,\ 0 \leq w_{id}, n \leq N,\ |F_1| = 2 \times (N + n - w_{id} - 1)\%N \\ \bot & \text{otherwise} \end{cases}$$

$$\begin{aligned} R \hookrightarrow \mathbb{R} \quad ::= & (\forall\, i \in \{0,\ldots,31\}.\, R(\mathtt{r}_i) = \mathbb{R}(\mathtt{r}_i)) \wedge (\forall\, \mathtt{sr} \neq \mathtt{wim}.R(\mathtt{sr}) = \mathbb{R}(\mathtt{sr})) \\ & \wedge R(\mathtt{n}) = \mathbb{R}(\mathtt{n}) \wedge R(\mathtt{z}) = \mathbb{R}(\mathtt{z}) \wedge R(\mathtt{c}) = \mathbb{R}(\mathtt{c}) \wedge R(\mathtt{v}) = \mathbb{R}(\mathtt{v}) \end{aligned}$$

$$\frac{\begin{array}{ccc} M_c = M_{\mathrm{ctx}} \uplus M_K & \mathrm{dom}(M_{\mathrm{ctx}}) = \mathsf{DomCtxM}(\mathsf{t}) & R(\%\mathtt{sp}) = (b,0) \\ \mathsf{ctxfm}(R,F) = F' & R(\%\mathtt{fp}) = (b',0) \quad (b',F',M_K) \Downarrow \mathbb{F} & R \hookrightarrow \mathbb{R} \end{array}}{(M_c,(R,F)) \Downarrow_{\mathsf{c}} (\mathsf{t},((\mathbb{R},\mathbb{F}),\mathtt{pc},\mathtt{npc}))}$$

$$\frac{M_1 \Downarrow_{\mathsf{r}} T_1 \quad M_2 \Downarrow_{\mathsf{r}} T_2}{M_1 \uplus M_2 \Downarrow_{\mathsf{r}} T_1 \uplus T_2} \qquad \frac{M \blacktriangleright_{\mathsf{t}} Q \quad (M,Q) \Downarrow_{\mathsf{c}} (\mathsf{t},\mathcal{K})}{M \Downarrow_{\mathsf{r}} \{\mathsf{t} \rightsquigarrow \mathcal{K}\}}$$

Fig.A8. Relation for Thread Pool and low-level Memory

of the contexts F (the pink part in the left side of the Fig. 12) are saved in register windows, and the other part of contexts M_K (the green part in the left side of the Fig. 12) are saved in stack frame in memory. However, in high-level state, they are abstracted as list named high-level frame list $\mathbb{F}$.

As shown in Fig. A7, if the low-level frame list F is nil and the memory is $\emptyset$, and the high-level frame list $\mathbb{F}$ is nil, it means there is no context stored. If the frame list is nil but the high-level frame list is $(b, \mathrm{fm}_1, \mathrm{fm}_2) :: \mathbb{F}$, it means that the contexts fm_1 and fm_2 are saved in stack frame in memory, whose block identifier is b. Here, we use "$\{(b, \mathrm{fm}_1, \mathrm{fm}_2)\}$" defined below to represent the part of memory saving fm_1 and fm_2. This memory contains only one block b.

$$\begin{aligned} \{(b,\mathrm{fm}_1,\mathrm{fm}_2)\} ::= & \{(b,0) \rightsquigarrow v_0, (b,4) \rightsquigarrow v_1, \ldots, (b,28) \rightsquigarrow v_7\} \\ & \uplus \{(b,32) \rightsquigarrow v'_0, (b,36) \rightsquigarrow v'_1, \ldots, (b,60) \rightsquigarrow v'_7\} \\ & \text{where } \mathrm{fm}_1 = [v_0,\ldots,v_7],\ \mathrm{fm}_2 = [v'_0,\ldots,v'_7]. \end{aligned}$$

If the frame list is $\mathrm{fm}_1 :: \mathrm{fm}_2 :: F$ and the high-level frame list is $(b', \mathrm{fm}'_1, \mathrm{fm}'_2) :: \mathbb{F}$, it means that the contexts fm_1 and fm_2 have not been saved in block b'. So, we require the contexts fm_1 and fm_2 saved in low-level frame list and the fm'_1 and fm'_2 saved in high-level frame list are equal. The block b' used to save fm_1 and fm_2 has not been used yet, so we don't care about its contents.

Relation for ThreadPool and low-level Memory. In high-level program, the thread local state of each thread is saved in a thread pool T. However, in low-level program, the local state of each thread is saved in memory (TCB and stack). For example, in Sec. 5, we introduce that the execution of the context switch module will save the register state of current thread into its TCB and stack in memory. So, the thread pool in high-level program can be viewed as an abstraction of low-level memory used to store the contexts of threads.

We define the relation between high-level thread pool and the memory used to save context in Fig. A8 formally. We use "$(M_c,(R,F)) \Downarrow_{\mathsf{c}} (\mathsf{t},((\mathbb{R},b,\mathbb{F}),\mathtt{pc},\mathtt{npc}))$" to represent the relation between the thread local states of *current thread* of low- and high-level program. The memory M_c owned the current thread t can be splitted into two parts M_{ctx} and M_{K}. The M_{ctx} are use the register file, whose domain is represented as $\mathsf{DomCtxM}(\mathsf{t}, b)$. It takes two arguments : the identifier t of the current thread and the block b of the stack frame at the top of the stack. Because

the context switch module may save the register file in TCB and the stack frame of the current procedure. The other part of the memory M_K is used to save the contexts of the previous procedures, which is abstracted as $\mathbb{F}$ in high-level program. We define $R \hookrightarrow \mathbb{R}$ to represent the relation between the register file R in low-level and $\mathbb{R}$ in high-level program. The operation $\mathsf{ctxfm}(R, F)$ is used to exact the prefix F_1 of the frame list F, which saves the contexts of the previous procedures. Suppoing the value of the `cwp` is w_{id}, meaning that the id of the current window is w_{id}, and the value of the `wim` is 2^n, meaning the id n register window is invalid. According to the introduction in Fig. 2.1, we usually set a window invalid to avoid over- and underflow of the register windows. So, we known that register windows id from $(w_{id}+1)\%N$ to $(n-1+N)\%N$ save the contexts of the previous procedures. So, we extract the contents F_1 of them from the whole frame list F.

We define "$M \Downarrow_{\mathsf{r}} \{\mathsf{t} \rightsquigarrow \mathcal{K}\}$" to represent the relation between the thread local states of *ready thread* of low- and high-level program. The operation "$M \blacktriangleright_{\mathsf{t}} Q$" means that we can restore the register state Q from memory M. When the context of the ready thread has been restored, we can establish a relation "$(M, Q) \Downarrow_{\mathsf{c}} (\mathsf{t}, \mathcal{K})$" between low- and high-level thread local states of thread t. Here, we don't represent the definitions of $\mathsf{DomCtxM}(\mathsf{t}, b)$ and $M \blacktriangleright_{\mathsf{t}} Q$ here, because their definitions are based on the implementation of the context switch routine in OS kernel. And the soundness of our extended program logic does not rely on their concrete definition.

Relation for Whole Program State. Finally, we introduce the state relation for whole program states between low- and high-level program below :

$$\frac{\begin{array}{c} M = M_c \uplus M_T \uplus \{\mathsf{TaskCur} \rightsquigarrow (\mathsf{t}, 0)\} \uplus M' \\ (M_c, Q) \Downarrow_{\mathsf{c}} (\mathsf{t}, \mathcal{K}) \qquad M_T \Downarrow_{\mathsf{r}} T\backslash\{\mathsf{t}\} \qquad D = \mathrm{nil} \end{array}}{(M, Q, D) \sim (T, \mathsf{t}, \mathcal{K}, M')}$$

D Application of Extended Program Logic : Verifying a Simplified Version of Context Switch Routine

In this section, we give a simplified version of context switch routine in Fig. A9. It reserves the main functionalities of the context switch routine introduced in Sec. 5, *e.g.*saving the contexts of currrent thread and restoring the new one. We omit some details like judging whether the current thread is a valid thread. We give a simple introduction to the function shown in Fig. A9, and show how to verify its correctness by applying our extended program logic for SPARCv8.

D.1 Simplified Context Switch Routine

At the entrance of the context switch rountine shown in Fig. A9, we first save the local and in registers into the stack in memory, and this part of the code is shown in Fig. A14(a). Then, as shown from line 2 to 6, we call the `reg_save` to store the out and global registers into the TCB of the current thread. As for the line 6 to 9, we get the identity of the current register window and the value of the `wim`. The block `Save_Usedwindows` (from line 10 to 20) saves the register windows (except the current one) into the stack of the current task in memory. It checks whether the previous window is valid. If it's valid, it uses the instruction `restore` to set the previous window as the current one, and save its contents (local and in registers) into stack in memory, then check the previous one continuously.

```
  SwitchEntry :
1     /* codes save the in and local registers of current window into stack frame */
2     set     TaskCur, %l1
3     ld      [%l1], %l1
4     call    reg_save
5     nop
6     mov     cwp, %g4
7     rd      wim, %g7
8     set     1, %g6
9     sll     %g6, %g4, %g4
  Save_Usedwindow :
10    sll     %g4, 1, %g5
11    srl     %g5, (OS_WINDOWS − 1), %g4
12    or      %g4, %g5, %g4
13    andcc   %g4, %g7, %g0
14    bne     Switch_NewContext
15    nop
16    restore %g0, %g0, %g0
17    /* codes save the in and local registers of current window into stack frame */
18    nop
19    jmp     Save_Usedwindow
20    nop
  Switch_NewContext :
21    set     TaskCur, %l0
22    set     TaskNew, %l1
23    ld      [%l1], %l1
24    st      %l1, [%l0]
25    call    reg_restore
26    nop
27    /* codes restore the in and local registers of current window from stack frame */
28    nop
29    retl
30    nop
```

Fig.A9. Main function of context switch routine

The block `Switch_NewContext` is responsible for restoring the context of the new task. From line 21 to 24, it sets the new task as the current one. Then, it calls function `reg_restore` (at line 25) to restore the out and global registers from the new task's TCB, and restores the local and in registers from the new task's stack in memory. The code about restoring the local and in registers of the new task from memory is shown in Fig. A14(b). The implementations of internal functions `reg_save` and `reg_restore` are omitted here because these two functions are taken from the OS kernal we verified and we can't show these part of codes according to confidentiality agreement, but we show their specifications in Fig. A12.

D.2 Specification of the Simplified Context Switch Routine

First, we define the abstract assembly primitive switch, which is already introduced in the Sec. 4.1.

$$\begin{aligned}\mathsf{switch} ::= \lambda\, \overline{v}, \mathbb{S}, \mathbb{S}'.\ \exists\, \mathtt{t}'.\ & M(\mathsf{TaskNew}) = (\mathtt{t}', 0) \wedge T(\mathtt{t}') = (\mathbb{Q}', \mathtt{pc}', \mathtt{npc}') \\ & \wedge T' = T\{\mathtt{t} \rightsquigarrow (\mathbb{Q}, \mathtt{pc}, \mathtt{npc})\} \wedge \mathtt{t} \neq \mathtt{t}' \wedge \overline{v} = \mathrm{nil} \\ \text{where } \mathbb{S} = (T, \mathtt{t}, (\mathbb{Q}, \mathtt{pc}, \mathtt{npc}), M),\ & \mathbb{S}' = (T', \mathtt{t}', (\mathbb{Q}', \mathtt{f}+8, \mathtt{f}+12), M), \mathtt{f} = \mathbb{Q}'.\mathbb{R}(\mathtt{r}_{15}).\end{aligned}$$

$$\mathsf{StkFrm}(b, \mathrm{fm}_1, \mathrm{fm}_2) ::= ((b,0) \Mapsto \mathrm{fm}_1[0]) * \ldots * ((b,28) \Mapsto \mathrm{fm}_1[7]) \\ *((b,32) \Mapsto \mathrm{fm}_2[0]) * \ldots * ((b,60) \Mapsto \mathrm{fm}_2[7])$$

$$\mathsf{RelStk}(b, F, \mathbb{F}) ::= \begin{cases} \mathsf{StkFrm}(b,\_,\_) * \mathsf{RelStk}(b', F', \mathbb{F}') & \textit{if } \mathrm{fm}_2[6] = (b', 0),\ F = \mathrm{fm}_1 :: \mathrm{fm}_2 :: F' \\ & \quad \mathbb{F} = (b, \mathrm{fm}_1, \mathrm{fm}_2) :: \mathbb{F}' \\ \mathsf{StkFrm}(b, \mathrm{fm}_1, \mathrm{fm}_2) * \mathsf{RelStk}(b', F', \mathbb{F}') & \textit{if } \mathrm{fm}_2[6] = (b', 0),\ F = \mathrm{nil} \\ & \quad \mathbb{F} = (b, \mathrm{fm}_1, \mathrm{fm}_2) :: \mathbb{F}' \\ \mathtt{Emp} & \textit{if } F = \mathrm{nil}, \mathbb{F} = \mathrm{nil} \\ \mathrm{false} & \mathrm{otherwise} \end{cases}$$

$$\mathsf{rRegs} ::= \mathtt{asr}_0 \mapsto \_ * \ldots \mathtt{asr}_{31} \mapsto \_ * \mathtt{Y} \mapsto \_$$

$$\mathsf{LRegs}(R) ::= \mathsf{global} \mapsto R(\mathsf{global}) * \mathsf{out} \mapsto R(\mathsf{out}) * \mathsf{local} \mapsto R(\mathsf{local}) * \mathsf{in} \mapsto R(\mathsf{in}) * \\ \mathtt{n} \mapsto R(\mathtt{n}) * \mathtt{z} \mapsto R(\mathtt{z}) * \mathtt{c} \mapsto R(\mathtt{z}) * \mathtt{v} \mapsto R(\mathtt{v}) * \mathsf{rRegs}$$

$$\mathsf{wfwin}(R, F) ::= (\mathtt{cwp} \mapsto (\!| R(\mathtt{cwp}), F |\!) * \mathtt{wim} \mapsto R(\mathtt{wim})) \wedge \mathsf{ctxfm}(R, F)$$

$$\mathsf{context}(\mathsf{t}, b, nst) ::= ((\mathsf{t}, \mathtt{G0\_OFFSET}) \Mapsto nst(\%\mathtt{g}_0)) * \ldots * ((\mathsf{t}, \mathtt{G7\_OFFSET}) \Mapsto nst(\%\mathtt{g}_7)) \\ *((\mathsf{t}, \mathtt{O0\_OFFSET}) \Mapsto nst(\%\mathtt{o}_0)) * \ldots * ((\mathsf{t}, \mathtt{O7\_OFFSET}) \Mapsto nst(\%\mathtt{o}_7)) \\ *((\mathsf{t}, \mathtt{N\_OFFSET}) \Mapsto nst(\mathtt{n})) * \ldots * ((\mathsf{t}, \mathtt{V\_OFFSET}) \Mapsto nst(\mathtt{v})) \\ *\mathsf{StkFrm}(b, nst[\mathsf{local}], nst[\mathsf{in}])$$

$$\mathsf{Env}(env) ::= \mathsf{LRegs}(R) * \mathsf{wfwin}(R, F) \qquad \text{where } env = (R, F)$$

$$\mathsf{CurT'}(\mathsf{t}_c, nst, env, \mathcal{K}) ::= (\mathsf{context}(\mathsf{t}_c, b, nst) * \mathsf{RelStk}(b', F, \mathbb{F}) * (\mathsf{t}_c \rightsquigarrow_{\mathsf{c}} \mathcal{K})) \\ \wedge R \hookrightarrow \mathbb{R} \wedge R(\%\mathtt{sp}) = (b, 0) \\ \text{where } env = (R, F),\ \mathcal{K} = ((\mathbb{R}, b, \mathbb{F}), \mathtt{pc}, \mathtt{npc}),\ R(\%\mathtt{fp}) = (b', 0),\ nst \in \mathrm{RegFile}$$

$$\mathsf{CurT}(\mathsf{t}_c, nst, env, \mathcal{K}) ::= (\mathsf{TaskCur} \mapsto (\mathsf{t}_c, 0)) * \mathsf{CurT'}(\mathsf{t}_c, nst, env, \mathcal{K})$$

$$\mathsf{RdyT}(\mathsf{t}_n, nst, \mathcal{K}) ::= (\mathsf{context}(\mathsf{t}_n, b, nst) * \mathsf{RelStk}(b', \mathrm{nil}, \mathbb{F}) * (\mathsf{t}_n \rightsquigarrow_{\mathsf{r}} \mathcal{K})) \\ \wedge nst \hookrightarrow \mathbb{R} \wedge R(\%\mathtt{sp}) = (b, 0) \\ \text{where } \mathcal{K} = ((\mathbb{R}, b, \mathbb{F}), \mathtt{pc}, \mathtt{npc}),\ nst(\%\mathtt{fp}) = (b', 0),\ nst \in \mathrm{RegFile}$$

$$\mathsf{p\_env}(env) ::= R \qquad \text{where } env = (R, F) \qquad\qquad l \Mapsto v ::= l \mapsto v * l \rightarrowtail v$$

Fig.A10. Auxiliary Definitions for Specification

Then we show the specification of the simplified context switch routine below, and some auxiliary definitions used in specification can be found in Fig. A10:

$$a_{pre}(\mathsf{t}_c, \mathsf{t}_n, env, nst, \mathcal{K}_c, \mathcal{K}_n) ::= \mathsf{Env}(env) * (\mathsf{TaskNew} \Mapsto (\mathsf{t}_n, 0) \wedge \mathsf{t}_c \neq \mathsf{t}_n) * \blacklozenge(10) * \\ \mathsf{CurT}(\mathsf{t}_c, \_, env, \mathcal{K}_c) * \mathsf{RdyT}(\mathsf{t}_n, nst, \mathcal{K}_n) * (\!| \mathsf{switch}(\mathrm{nil}) |\!)$$

$$a_{post}(\mathsf{t}_c, \mathsf{t}_n, env, nst, \mathcal{K}_c, \mathcal{K}_n) ::= \exists\, env', \mathcal{K}'.\, \mathsf{Env}(env') * (\mathsf{TaskNew} \Mapsto (\mathsf{t}_n, 0) \wedge \mathsf{t}_c \neq \mathsf{t}_n) * \\ \mathsf{CurT}(\mathsf{t}_n, nst, env', \mathcal{K}') * \mathsf{RdyT}(\mathsf{t}_c, \mathsf{p\_env}(env), \mathcal{K}_c) * (\!| \bot |\!)$$

Note that the execution of context switch routine will call function `reg_save`, `reg_restore`, and `window_restore` once, and call function and jump to block `save_usedwindow` no more than 8 times separately, because the number of the register windows is 8. So, assigning 10 tokens to the precondition of the context switch routine is enough. According to the logic rules of extended program logic shown in Fig. 19, we need to check whether the specification of context switch rountine is well-defined.

Lemma 10. $\mathsf{wdSpec}(a_{pre}, a_{post}, \mathsf{switch})$.

Proof. We unfold $\mathsf{wdSpec}(a_{pre}, a_{post}, \mathsf{switch})$ by Def. 2, and we need to prove three properties about the specification and abstract assembly primitive switch.

1. for any $\overline{v}, \mathbb{S}, \mathbb{S}', \mathbb{S}_r$. if $\mathsf{switch}(\overline{v})(\mathbb{S})(\mathbb{S}')$, and $\mathbb{S} \perp \mathbb{S}_r$, then the following holds :

   - $\mathbb{S}'.\mathcal{K}.\mathtt{pc} = \mathtt{f} + 8$, $\mathbb{S}'.\mathcal{K}.\mathtt{npc} = \mathtt{f} + 12$ (where $\mathbb{S}'.\mathcal{K}.\mathbb{Q}.\mathbb{R}(\mathtt{r}_{15}) = \mathtt{f}$);
   - there exists $\mathbb{S}'', \mathbb{S}'_r$, $\mathsf{switch}(\overline{v})(\mathbb{S} \uplus \mathbb{S}_r)(\mathbb{S}'')$, $\mathbb{S}'' = \mathbb{S}' \uplus \mathbb{S}'_r$, and $\mathbb{S}_r.T = \mathbb{S}'_r.T$, $\mathbb{S}_r.M = \mathbb{S}'_r.M$;

   The correctness proof of this property can be achieved directly from the definition of the switch. The definition of the switch requires that the program counters pc and npc are equal to f+8 and f+12, where f is contained in $\mathtt{r}_{15}$ register after the execution of switch. The execution of switch only accesses the current thread t and new t' (stored in TaskNew in memory) in thread pool and the location TaskNew in memory. So the threads and memory described in $\mathbb{S}_r$ remains unchanged.

   We prove the correctness of this property formally below. We unfold $\mathsf{switch}(\overline{v})(\mathbb{S})(\mathbb{S}')$ and $\mathbb{S} \perp \mathbb{S}_r$ according to their definitions and get the following hold: (let $\mathbb{S} = (T, \mathsf{t}, (\mathbb{Q}, \mathtt{pc}, \mathtt{npc}), M)$ and $\mathbb{S}_r = (T_r, \mathsf{t}_r, \mathcal{K}_r, M_r)$)

   $$M(\mathsf{TaskNew}) = (\mathsf{t}', 0) \tag{23}$$
   $$T(\mathsf{t}') = (\mathbb{Q}', \mathtt{pc}', \mathtt{npc}') \tag{24}$$
   $$T' = T\{\mathsf{t} \rightsquigarrow \{\mathbb{Q}, \mathtt{pc}, \mathtt{npc}\}\} \tag{25}$$
   $$\mathsf{t} \neq \mathsf{t}', \overline{v} = \mathrm{nil} \tag{26}$$
   $$\mathbb{S}' = (T', \mathsf{t}', (\mathbb{Q}', \mathtt{f}+8, \mathtt{f}+12), M), \mathtt{f} = \mathbb{Q}'.\mathbb{R}(\mathtt{r}_{15}) \tag{27}$$

   $$T \perp T_r,\ M \perp M_r \tag{28}$$
   $$\mathsf{t}_r = \mathsf{t},\ \mathcal{K}_r = (\mathbb{Q}, \mathtt{pc}, \mathtt{npc}) \tag{29}$$

   We can prove that there exists $\mathbb{S}'' = (T' \uplus T_r, \mathsf{t}', (\mathbb{Q}', \mathtt{f}+8, \mathtt{f}+12), M \uplus M_r)$, and $\mathbb{S}'_r = (T_r, \mathsf{t}', (\mathbb{Q}', \mathtt{f}+8, \mathtt{f}+12), M_r)$, such that $\mathsf{switch}(\overline{v})(\mathbb{S} \uplus \mathbb{S}_r)(\mathbb{S}'')$, $\mathbb{S}'' = \mathbb{S}' \uplus \mathbb{S}'_r$, and $\mathbb{S}_r.T = \mathbb{S}'_r.T$, $\mathbb{S}_r.M = \mathbb{S}'_r.M$.

2. for any $\mathsf{t}_c, \mathsf{t}_n, \mathit{env}, \mathit{nst}, \mathcal{K}_c, \mathcal{K}_n$,

   - $a_{pre}(\mathsf{t}_c, \mathsf{t}_n, \mathit{env}, \mathit{nst}, \mathcal{K}_c, \mathcal{K}_n) \Longrightarrow (\!|\mathsf{switch}|\!) * \mathrm{true}$;
   - $a_{post}(\mathsf{t}_c, \mathsf{t}_n, \mathit{env}, \mathit{nst}, \mathcal{K}_c, \mathcal{K}_n) \Longrightarrow (\!|\bot|\!) * \mathrm{true}$;

   According the definition of a_{pre} and a_{post}, this property's proof is trivial.

3. for any $\overline{v}, S, \mathbb{S}$, if $(S, \mathbb{S}, \_, \_) \in \mathsf{INV}(\mathsf{switch}(\overline{v}), \overline{v})$, then there exists $\mathsf{t}_c, \mathsf{t}_n, \mathit{env}, \mathit{nst}, \mathcal{K}_c, \mathcal{K}_n, \mathbb{p}_r$ and w, such that:

   - $(S, \mathbb{S}, \mathsf{switch}(\overline{v}), w) \models a_{pre}(\mathsf{t}_c, \mathsf{t}_n, \mathit{env}, \mathit{nst}, \mathcal{K}_c, \mathcal{K}_n) * \mathbb{p}_r$;
   - $a_{post}(\mathsf{t}_c, \mathsf{t}_n, \mathit{env}, \mathit{nst}, \mathcal{K}_c, \mathcal{K}_n) * \mathbb{p}_r \Longrightarrow \mathsf{INV}(\bot, \_)$;
   - $\mathsf{Sta}(\mathsf{switch}(\overline{v}), \mathbb{p}_r)$.

$$\mathsf{Mem}(M) \quad ::= \begin{cases} \mathtt{Emp} & \mathit{if}\ M = \emptyset \\ (l \Mapsto v) * (l \rightarrowtail v) & \mathit{if}\ M = \{l \rightsquigarrow v\} \\ \exists\, M_1, M_2.\, \mathsf{Mem}(M_1) * \mathsf{Mem}(M_2) & \text{otherwise} \end{cases}$$

$$\mathsf{RdyTs}(T) \quad ::= \begin{cases} \mathtt{Emp} & \mathit{if}\ T = \emptyset \\ \mathsf{RdyT}(\mathtt{t}, \_, \mathcal{K}) & \mathit{if}\ T = \{\mathtt{t} \rightsquigarrow \mathcal{K}\} \\ \exists\, T_1, T_2.\, \mathsf{RdyTs}(T_1) * \mathsf{RdyTs}(T_2) & \text{otherwise} \end{cases}$$

$$\begin{aligned} M \blacktriangleright_{\mathtt{t}} (R, F) ::= {} & \exists\, b.\, R(\%\mathtt{g}_0) = M(\mathtt{t}, \mathtt{G0\_OFFSET}) \wedge \ldots \wedge R(\%\mathtt{g}_7) = M(\mathtt{t}, \mathtt{G7\_OFFSET}) \\ & \wedge R(\%\mathtt{o}_0) = M(\mathtt{t}, \mathtt{O0\_OFFSET}) \wedge \ldots \wedge R(\%\mathtt{o}_7) = M(\mathtt{t}, \mathtt{O7\_OFFSET}) \\ & \wedge R(\mathtt{n}) = M(\mathtt{t}, \mathtt{N\_OFFSET}) \wedge \ldots \wedge R(\mathtt{v}) = M(\mathtt{t}, \mathtt{V\_OFFSET}) \\ & \wedge R(\%\mathtt{l}_0) = M(b, 0) \wedge \ldots \wedge R(\%\mathtt{l}_7) = M(b, 28) \\ & \wedge R(\%\mathtt{i}_0) = M(b, 32) \wedge \ldots \wedge R(\%\mathtt{i}_7) = M(b, 60) \wedge R(\%\mathtt{sp}) = (b, 0) \\ & \wedge (\exists\, w_{id}, n.\, R(\mathtt{cwp}) = w_{id} \wedge R(\mathtt{wim}) = 2^n \wedge \mathbf{prev\_cwp}(w_{id}) = n) \end{aligned}$$

Fig.A11. Auxiliary Definitions About Frame Assertion

The key to prove this case is to find $\mathtt{t}_c$, $\mathtt{t}_n$, *env*, *nst*, $\mathcal{K}_c$, $\mathcal{K}_n$, $\mathtt{p}_r$ and w. Because we have $(S, \mathbb{S}, \_, \_) \in \mathsf{INV}(\mathsf{switch}(\overline{v}), \overline{v})$, we know that there exists a ready thread $\mathtt{t}'$, a prefix of the frame list F' and a register state Q', where $\mathbb{S}.T(\mathtt{t}') = \mathcal{K}'$, $\mathtt{t} \neq \mathtt{t}'$, $\mathbb{S}.M(\mathsf{TaskNew}) = (\mathtt{t}', 0)$, $\mathsf{ctxfm}(S.Q) = F'$ and $S.M \blacktriangleright_{\mathtt{t}'} Q'$ hold. And we require $\mathtt{t}_c = \mathbb{S}.\mathtt{t}$, $\mathtt{t}_n = \mathtt{t}'$, $env = (S.Q.R, F')$, $nst = Q'.R$, $w = 10$, and $\mathtt{p}_r = \exists\, M, T.\, \mathsf{Mem}(M) * \mathsf{RdyTs}(T)$. Then, we can finish the proof.

We prove the correctness of this property formally below. We first unfold $(S, \mathbb{S}, \_, \_) \in \mathsf{INV}(\mathsf{switch}(\overline{v}), v)$ and get the following holds:

$$S \sim \mathbb{S} \tag{30}$$

$$\exists\, \mathbb{S}'.\, (\mathsf{switch}(\overline{v}), \mathbb{S}) \dashrightarrow^* (\bot, \mathbb{S}') \tag{31}$$

$$\mathbf{args}(\mathbb{S}.\mathcal{K}.\mathbb{Q}, \mathbb{S}.M, \overline{v}) \tag{32}$$

Let $\mathbb{S} = (T, \mathtt{t}, (\mathbb{Q}, \mathtt{pc}, \mathtt{npc}), M')$, $S = (M, Q, D)$. We first unfold $S \sim \mathbb{S}$ (30) according to its definition.

$$M = M_c \uplus M_T \uplus \{\mathsf{TaskCur} \rightsquigarrow (\mathtt{t}, 0)\} \uplus M' \tag{33}$$

$$(M_c, Q) \Downarrow_{\mathsf{c}} (\mathtt{t}, (\mathbb{Q}, \mathtt{pc}, \mathtt{npc})) \tag{34}$$

$$M_T \Downarrow_{\mathsf{r}} T \backslash \{\mathtt{t}\} \tag{35}$$

$$D = \text{nil} \tag{36}$$

We unfold the execution of switch (31) and get that there exists $\mathtt{t}'$, such that:

$$M'(\mathsf{TaskNew}) = (\mathtt{t}', 0) \tag{37}$$

$$T(\mathtt{t}') = (\mathbb{Q}', \mathtt{pc}', \mathtt{npc}') \tag{38}$$

$$T' = T\{\mathtt{t} \rightsquigarrow \{\mathbb{Q}, \mathtt{pc}, \mathtt{npc}\}\} \tag{39}$$

$$\mathtt{t} \neq \mathtt{t}', \overline{v} = \text{nil} \tag{40}$$

$$\mathbb{S}' = (T', \mathtt{t}', (\mathbb{Q}', \mathtt{f}+8, \mathtt{f}+12), M), \mathtt{f} = \mathbb{Q}'.\mathbb{R}(\mathtt{r}_{15}) \tag{41}$$

From (40) and (35), we know that $\mathtt{t}' \in \text{dom}(T \backslash \{\mathtt{t}\})$ and we can split M_T as two parts: $M_{\mathtt{t}'}$ storing the contexts of the new task $\mathtt{t}'$, and M'_T storing the contexts of the rest of the ready threads.

$$M_T = M_{\mathtt{t}'} \uplus M'_T \tag{42}$$

$$M_{\mathtt{t}'} \blacktriangleright_{\mathtt{t}'} Q' \tag{43}$$

$$(M_{\mathtt{t}'}, Q') \Downarrow_{\mathsf{c}} (\mathtt{t}', (\mathbb{Q}', \mathtt{pc}', \mathtt{npc}')) \tag{44}$$

$$M'_T \Downarrow_{\mathsf{r}} T \backslash \{\mathtt{t}, \mathtt{t}'\} \tag{45}$$

Then, we can prove that there exists $\mathtt{t}_c = \mathtt{t}$, $\mathtt{t}_n = \mathtt{t}'$, $env = (Q.R, F')$ (where $F' = \mathsf{ctxfm}(Q)$), $nst = \mathbb{Q}'.R$, $w = 10$, $\mathcal{K}_c = (\mathbb{Q}, \mathtt{pc}, \mathtt{npc})$, $\mathcal{K}_n = (\mathbb{Q}', \mathtt{pc}', \mathtt{npc}')$ and $\mathbb{p}_r = \exists\, M, T.\, \mathsf{Mem}(M) * \mathsf{RdyTs}(T) * \mathsf{rRegs}$, such that

- $(S, \mathbb{S}, \mathsf{switch}(\text{nil}), w) \models a_{pre}(\mathtt{t}_c, \mathtt{t}_n, env, nst, \mathcal{K}_c, \mathcal{K}_n) * \mathbb{p}_r$ holds, because we can prove that there exists S_1, S_2, $\mathbb{S}_1$ and $\mathbb{S}_2$, such that:

$$S = S_1 \uplus S_2 \tag{46}$$

$$\mathbb{S} = \mathbb{S}_1 \uplus \mathbb{S}_2 \tag{47}$$

$$S_1 = (M_c \uplus M_{\mathtt{t}'} \uplus \{\mathsf{TaskCur} \rightsquigarrow (\mathtt{t}, 0), \mathsf{TaskNew} \rightsquigarrow (\mathtt{t}', 0)\}, Q, \text{nil}) \tag{48}$$

$$S_2 = (M'_T \uplus (M' \backslash \{\mathsf{TaskNew}\}), (\emptyset, Q.F), \text{nil}) \tag{49}$$

$$\mathbb{S}_1 = (\{\mathtt{t} \rightsquigarrow T(\mathtt{t}), \mathtt{t}' \rightsquigarrow (\mathbb{Q}', \mathtt{pc}', \mathtt{npc}')\}, \mathtt{t}, (\mathbb{Q}, \mathtt{pc}, \mathtt{npc}), \{\mathsf{TaskNew} \rightsquigarrow (\mathtt{t}', 0)\}) \tag{50}$$

$$\mathbb{S}_2 = (T \backslash \{\mathtt{t}, \mathtt{t}'\}, \mathtt{t}, (\mathbb{Q}, \mathtt{pc}, \mathtt{npc}), M' \backslash \{\mathsf{TaskNew}\}) \tag{51}$$

$$(S_1, \mathbb{S}_1, \mathsf{switch}(\text{nil}), w) \models a_{pre}(\mathtt{t}_c, \mathtt{t}_n, env, nst, \mathcal{K}_c, \mathcal{K}_n) \tag{52}$$

$$(S_2, \mathbb{S}_2, \mathsf{switch}(\text{nil}), w) \models \mathbb{p}_r \tag{53}$$

- $a_{post}(\mathtt{t}_c, \mathtt{t}_n, env, nst, \mathcal{K}_c, \mathcal{K}_n) * \mathbb{p}_r \Longrightarrow \mathsf{INV}(\bot, \_)$.
  Supposing $(S', \mathbb{S}', A, w) \models a_{post}(\mathtt{t}_c, \mathtt{t}_n, env, nst, \mathcal{K}_c, \mathcal{K}_n) * \mathbb{p}_r$, we get there exists S'_1, S'_2, $\mathbb{S}'_1$ and $\mathbb{S}'_2$, such that:

$$S' = S'_1 \uplus S'_2 \tag{54}$$

$$\mathbb{S}' = \mathbb{S}'_1 \uplus \mathbb{S}'_2 \tag{55}$$

$$(S'_1, \mathbb{S}'_1, A, w) \models a_{post}(\mathtt{t}_c, \mathtt{t}_n, env, nst, \mathcal{K}_c, \mathcal{K}_n) \tag{56}$$

$$(S'_2, \mathbb{S}'_2, A, w) \models \mathbb{p}_r \tag{57}$$

Then, according to the definition of a_{post}, we get there exists M'_n, M'_c, Q'_c, Q', $\mathcal{K}'_n$ and $\mathcal{K}'$, such that:

$$S'_1 = (M'_n \uplus M'_{\mathtt{t}} \uplus \{\mathsf{TaskCur} \rightsquigarrow (\mathtt{t}_n, 0), \mathsf{TaskNew} \rightsquigarrow (\mathtt{t}_n, 0)\}, Q', \text{nil}) \tag{58}$$

$$\mathbb{S}'_1 = (\{\mathtt{t}_n \rightsquigarrow \mathcal{K}'_n, \mathtt{t}_c \rightsquigarrow \mathcal{K}_c\}, \mathtt{t}_n, \mathcal{K}', \{\mathsf{TaskNew} \rightsquigarrow (\mathtt{t}_n, 0)\}) \tag{59}$$

$$(M'_n, Q'_n) \Downarrow_{\mathsf{c}} (\mathtt{t}_n, \mathcal{K}') \tag{60}$$

$$M'_c \blacktriangleright_{\mathtt{t}_c} Q'_c,\ (M'_c, Q'_c) \Downarrow_{\mathsf{c}} (\mathtt{t}_c, \mathcal{K}_c) \tag{61}$$

$$A = \bot \tag{62}$$

And according to the definition of $\mathtt{p}_r$, we get there exists M'_{T_r}, M'_r and T'_r such that:

$$S'_2 = (M'_{T_r} \uplus M'_r, (\mathrm{nil}, Q'.F), \mathrm{nil}) \tag{63}$$

$$\mathbb{S}'_2 = (T'_r, \mathtt{t}_n, \mathcal{K}', M'_r) \tag{64}$$

$$M'_{T_r} \Downarrow_{\mathsf{r}} T'_r \tag{65}$$

Then, we get the following hold:

$$S.M = M'_n \uplus (M'_c \uplus M'_{T_r}) \uplus \{\mathsf{TaskCur} \rightsquigarrow (\mathtt{t}_n, 0), \mathsf{TaskNew} \rightsquigarrow (\mathtt{t}_n, 0)\} \uplus M'_r \tag{66}$$

$$(M'_n, Q'_n) \Downarrow_{\mathsf{c}} (\mathtt{t}_n, \mathcal{K}') \tag{67}$$

$$M'_c \uplus M'_{T_r} \Downarrow_{\mathsf{r}} \{\mathtt{t}_c \rightsquigarrow \mathcal{K}_c\} \uplus T'_r \tag{68}$$

Thus, we get $S' \sim \mathbb{S}'$, and $\mathsf{INV}(\bot, \_)$ holds.

□

We use $C_{\texttt{switch}}$ to represent the code heap that stores the code of context switch rountine shown in Fig. A9. The specifications of the internal function can be found in Fig. A12. The function `reg_save` is responsible for saving the local, in and integer condition code fields `n`, `z`, `c` and `v` registers into TCB in memory. And the function `reg_restore` does the reverse of `reg_save`, restoring local, in and integer condition code fields registers from the TCB in memory. The specification of code block `Save_Usedwindows` is a little complicated. We can find its implementation is a loop, which checks whether the previous window is valid and saving the contents of the valid previous window until the previous one is invalid. We need to define the loop invariant I here.

Codes about saving local and in registers into stack in memory and restoring local and in registers from stack in memory are not implemented as functions. However, we still give the specifications of this part of codes shown as `window_save` and `window_restore` in Fig. A12, in order to allow readers to better understand the functionalities of this part of codes.

We define the code heap specification Ψ below, which is the collection of specifications of code blocks in $C_{\texttt{switch}}$.

$$\begin{aligned}\Psi ::= \{&\texttt{SwitchEntry} \rightsquigarrow (a_{pre}, a_{post}),\ \texttt{reg\_save} \rightsquigarrow (\mathtt{fp}_{rs}, \mathtt{fq}_{rs}),\ \texttt{reg\_restore} \rightsquigarrow (\mathtt{fp}_{rr}, \mathtt{fq}_{rr}),\\ &\texttt{Save\_Usedwindows} \rightsquigarrow (\mathtt{fp}_{su}, \mathtt{fq}_{su}),\ \texttt{Switch\_NewContext} \rightsquigarrow (\mathtt{fp}_{sn}, \mathtt{fq}_{sn}))\}\end{aligned} \tag{69}$$

Lemma 11. $\vdash C_{\texttt{switch}} : \Psi$

Proof. The code heap specification Ψ have been defined in (69). We unfold $\vdash C_{\texttt{switch}} : \Psi$ according to its definition (in Fig. 19). And we need to prove that, for any ι_1, ι_2, ι_3, ι_4 and ι_5, the following hold:

$$\Psi \vdash \{(a_{pre}\ \iota_1, a_{post}\ \iota_1)\}\ \texttt{SwitchEntry} : C_{\texttt{switch}}[\texttt{SwitchEntry}] \tag{70}$$

Loop invariant I :

$\mathsf{wptr}(R_0) ::= (R_0(\%\mathtt{g}_7) = R_0(\mathtt{wim})) \wedge$
$((R_0(\%\mathtt{g}_4) = (1 \gg R_0(\mathtt{cwp}))) \vee (R_0(\%\mathtt{g}_4) = ((1 \gg R_0(\mathtt{cwp}))|(1 \gg (R_0(\mathtt{cwp})+8)))))$

$\mathsf{linkF}((b_1,\mathbb{F}_1),(b_2,\mathbb{F}_2),\mathbb{F}) ::= \mathbb{F}_1 \cdot \mathbb{F}_2 = \mathbb{F} \wedge (\mathbb{F}_1 = \mathrm{nil} \to b_1 = b_2)$
$(\forall b, \mathrm{fm}_1, \mathrm{fm}_2, \mathbb{F}'.\, \mathbb{F}_1 = (b, \mathrm{fm}_1, \mathrm{fm}_2) :: \mathbb{F}' \to \mathrm{fm}_2[6] = (b_2, 0))$

$I(\mathsf{t}_c, R, \mathcal{K}_c) ::= \exists\, R_0, F_0.\, (\mathsf{Env}(R_0, F_0) \wedge \mathsf{wptr}(R_0)) * \blacklozenge(|F_0| + 2)$
$*((\mathsf{TaskCur} \mapsto (\mathsf{t}_c, 0) * \mathsf{context}(\mathsf{t}_c, b, R) * (\mathsf{t}_c \rightsquigarrow_{\mathsf{c}} \mathcal{K}_c)) \wedge R \hookrightarrow \mathbb{R})$
$*(\exists\, b'', \mathbb{F}_1, \mathbb{F}_2.\, (\mathsf{RelStk}(b', \mathrm{nil}, \mathbb{F}_1) * \mathsf{RelStk}(b'', F_0, \mathbb{F}_2)) \wedge R_0(\%\mathtt{sp}) = (b'', 0)$
$\wedge \mathsf{linkF}((b', \mathbb{F}_1), (b'', \mathbb{F}_2), \mathbb{F}))$

where $\mathcal{K}_c = (\mathbb{R}, b, \mathbb{F})$, $R(\%\mathtt{sp}) = (b, 0)$, and $R(\%\mathtt{fp}) = (b', 0)$

`reg_save` : $(\iota = (\mathsf{t}, \mathcal{K}, R, F, A, nst))$

$\mathtt{fp}_{rs}\ \iota ::= \mathsf{Env}(R, F) * \mathsf{context}(\mathsf{t}, b, nst) * (\mathsf{t} \rightsquigarrow_{\mathsf{c}} \mathcal{K}) * (\!|A|\!)$

$\mathtt{fq}_{rs}\ \iota ::= (\exists\, nst'.\, \mathsf{Env}(R, F) * \mathsf{context}(\mathsf{t}, b, nst')) * (\mathsf{t} \rightsquigarrow_{\mathsf{c}} \mathcal{K}) * (\!|A|\!)$
$\wedge\, nst' = nst\{\mathsf{global} \rightsquigarrow R(\mathsf{global}), \mathsf{out} \rightsquigarrow R(\mathsf{out}), \mathtt{n} \rightsquigarrow R(\mathtt{n}), \ldots, \mathtt{v} \rightsquigarrow R(\mathtt{v})\}$

`reg_restore` : $(\iota = (\mathsf{t}, \mathcal{K}, R, F, b, nst))$

$\mathtt{fp}_{rr}\ \iota ::= \mathsf{Env}(R, F) * \mathsf{context}(\mathsf{t}, b, nst) * (\mathsf{t} \rightsquigarrow_{\mathsf{c}} \mathcal{K}) * (\!|A|\!)$

$\mathtt{fq}_{rr}\ \iota ::= \exists\, R'.\, (\mathsf{Env}(R', F) * \mathsf{context}(\mathsf{t}, b, nst) * (\mathsf{t} \rightsquigarrow_{\mathsf{c}} \mathcal{K}) * (\!|A|\!)$
$\wedge\, R' = R\{\mathsf{global} \rightsquigarrow nst(\mathsf{global}), \mathsf{out} \rightsquigarrow nst(\mathsf{out}), \mathtt{n} \rightsquigarrow nst(\mathtt{n}), \ldots, \mathtt{v} \rightsquigarrow nst(\mathtt{v})\}$

`Save_Usedwindows` : $(\iota = (\mathsf{t}_c, \mathsf{t}_n, \mathcal{K}_c, \mathcal{K}_n, \mathcal{K}_c, \mathcal{K}_n, nst))$

$\mathtt{fp}_{su}\ \iota ::= I(\mathsf{t}_c, R, \mathcal{K}_c) * (\mathsf{TaskNew} \Mapsto (\mathsf{t}_n, 0) \wedge \mathsf{t}_c \neq \mathsf{t}_n) * \mathsf{RdyT}(\mathsf{t}_n, nst, \mathcal{K}_n) * (\!|\, \mathsf{switch}(\mathrm{nil}) \,|\!)$

$\mathtt{fq}_{su}\ \iota ::= \exists\, \mathcal{K}'.\, \mathsf{Env}(nst, \mathrm{nil}) * (\mathsf{TaskNew} \Mapsto (\mathsf{t}_n, 0) \wedge \mathsf{t}_c \neq \mathsf{t}_n) *$
$\mathsf{CurT}(\mathsf{t}_n, nst, (nst, \mathrm{nil}), \mathcal{K}') * \mathsf{RdyT}(\mathsf{t}_c, R, \mathcal{K}_c) * (\!|\, \bot \,|\!)$

`Switch_NewContext` $(\iota = (\mathsf{t}_c, \mathsf{t}_n, \mathcal{K}_c, \mathcal{K}_n, R, nst))$

$\mathtt{fp}_{sn}\ \iota ::= \exists\, R_0.\, \mathsf{Env}((R_0, \mathrm{nil})) * (\mathsf{TaskNew} \Mapsto (\mathsf{t}_n, 0) \wedge \mathsf{t}_c \neq \mathsf{t}_n) * \blacklozenge(1) *$
$\mathsf{CurT}(\mathsf{t}_c, R, (R, \mathrm{nil}), \mathcal{K}_c) * \mathsf{RdyT}(\mathsf{t}_n, nst, \mathcal{K}_n) * (\!|\, \mathsf{switch}(\mathrm{nil}) \,|\!)$

$\mathtt{fq}_{sn}\ \iota ::= \exists\, \mathcal{K}'.\, \mathsf{Env}(nst, \mathrm{nil}) * (\mathsf{TaskNew} \Mapsto (\mathsf{t}_n, 0) \wedge \mathsf{t}_c \neq \mathsf{t}_n) *$
$\mathsf{CurT}(\mathsf{t}_n, nst, (nst, \mathrm{nil}), \mathcal{K}') * \mathsf{RdyT}(\mathsf{t}_c, R, \mathcal{K}_c) * (\!|\, \bot \,|\!)$

`window_save` : $(\iota = (\mathsf{t}, \mathcal{K}, R, F, A, b))$

$\mathtt{fp}_{ws}\ \iota ::= (\mathsf{Env}(R, F) * \mathsf{StkFrm}(b, \_, \_) * (\mathsf{t} \rightsquigarrow_{\mathsf{c}} \mathcal{K}) * (\!|A|\!)) \wedge R(\%\mathtt{sp}) = (b, 0)$

$\mathtt{fq}_{ws}\ \iota ::= (\mathsf{Env}(R, F) * \mathsf{StkFrm}(b, R(\mathsf{local}), R(\mathsf{in})) * (\mathsf{t} \rightsquigarrow_{\mathsf{c}} \mathcal{K}) * (\!|A|\!)) \wedge R(\%\mathtt{sp}) = (b, 0)$

`window_restore` : $(\iota = (\mathsf{t}, \mathcal{K}, R, F, A, b, \mathrm{fm}_1, \mathrm{fm}_2))$

$\mathtt{fp}_{wr}\ \iota ::= (\mathsf{Env}(R, F) * \mathsf{StkFrm}(b, \mathrm{fm}_1, \mathrm{fm}_2) * (\mathsf{t} \rightsquigarrow_{\mathsf{c}} \mathcal{K}) * (\!|A|\!)) \wedge R(\%\mathtt{sp}) = (b, 0)$

$\mathtt{fq}_{wr}\ \iota ::= (\mathsf{Env}(R\{\mathsf{local} \rightsquigarrow \mathrm{fm}_1, \mathsf{in} \rightsquigarrow \mathrm{fm}_2\}, F)\ * \mathsf{StkFrm}(b, \mathrm{fm}_1, \mathrm{fm}_2) * (\mathsf{t} \rightsquigarrow_{\mathsf{c}} \mathcal{K}) * (\!|A|\!)) \wedge R(\%\mathtt{sp}) = (b, 0)$

Fig.A12. Specifications of Internal Functions

$$\Psi \vdash \{(\mathtt{fp}_{rs}\ \iota_2, \mathtt{fq}_{rs}\ \iota_2)\}\ \mathtt{reg\_save} : C_{\mathtt{switch}}[\mathtt{reg\_save}] \tag{71}$$

$$\Psi \vdash \{(\mathtt{fp}_{rr}\ \iota_3, \mathtt{fq}_{rr}\ \iota_3)\}\ \mathtt{reg\_restore} : C_{\mathtt{switch}}[\mathtt{reg\_restore}] \tag{72}$$

$$\Psi \vdash \{(\mathtt{fp}_{su}\ \iota_4, \mathtt{fq}_{su}\ \iota_4)\}\ \mathtt{Save\_Usedwindows} : C_{\mathtt{switch}}[\mathtt{Save\_Usedwindows}] \tag{73}$$

$$\Psi \vdash \{(\mathtt{fp}_{sn}\ \iota_5, \mathtt{fq}_{sn}\ \iota_5)\}\ \mathtt{Switch\_NewContext} : C_{\mathtt{switch}}[\mathtt{Switch\_NewContext}] \tag{74}$$

We need to prove that each code block is well-defined ((70) - (74)). Here, we do not show the details about verifying each code block respectively. We just give a proof sketch of the verifying of the main function, which can be found in Fig. A13.

Supposing in the inital state (described as assertion marked ①), the register file is R, and the part of the

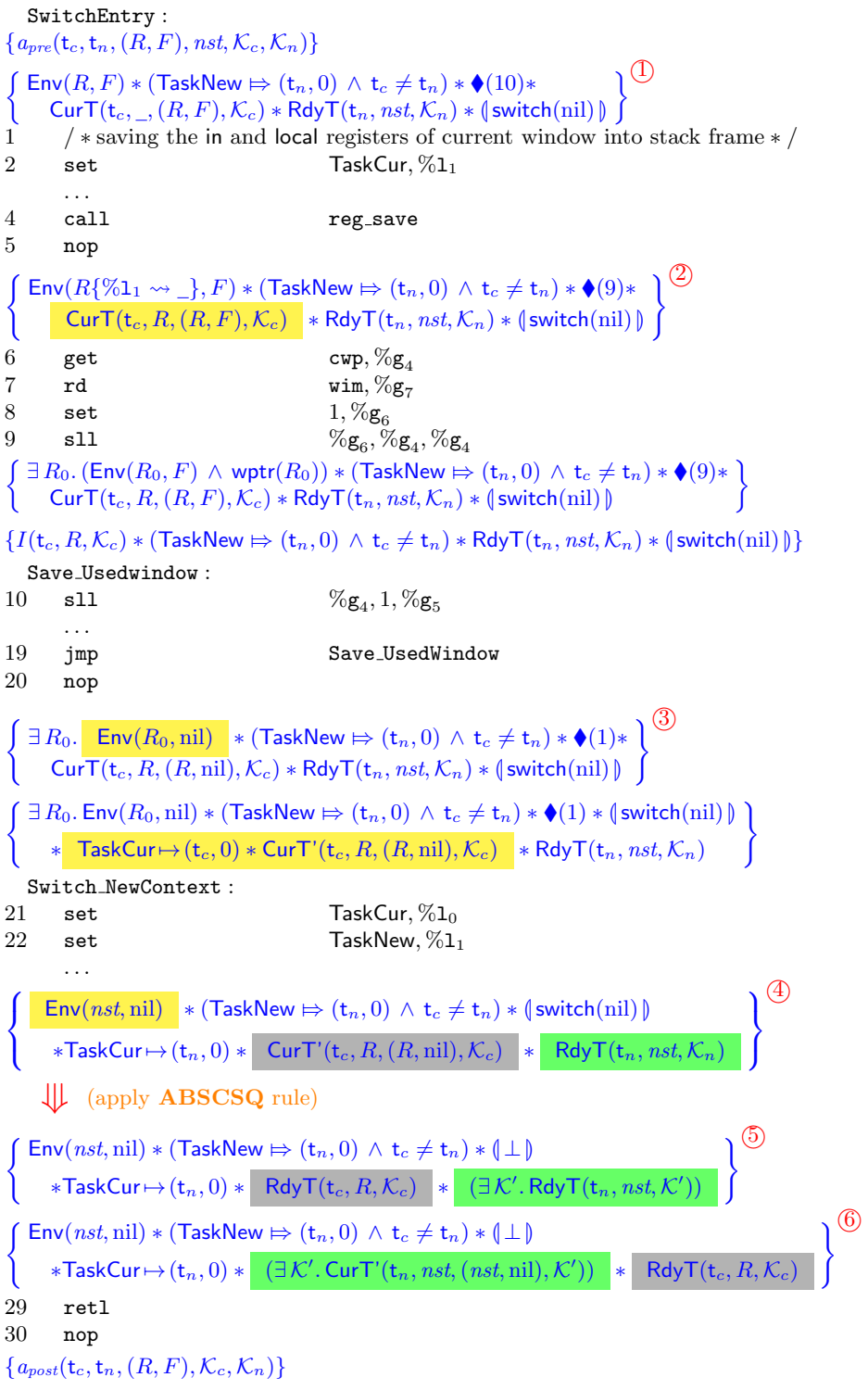


Fig.A13. Proof Sketch of the Context Switch Routine

```
st %l0, [%sp + L0_OFFSET]
st %l1, [%sp + L1_OFFSET]
st %l2, [%sp + L2_OFFSET]
st %l3, [%sp + L3_OFFSET]
st %l4, [%sp + L4_OFFSET]
st %l5, [%sp + L5_OFFSET]
st %l6, [%sp + L6_OFFSET]
st %l7, [%sp + L7_OFFSET]
st %i0, [%sp + I0_OFFSET]
st %i1, [%sp + I1_OFFSET]
st %i2, [%sp + I2_OFFSET]
st %i3, [%sp + I3_OFFSET]
st %i4, [%sp + I4_OFFSET]
st %i5, [%sp + I5_OFFSET]
st %i6, [%sp + I6_OFFSET]
st %i7, [%sp + I7_OFFSET]
```

(a) Save local and out into memory

```
ld [%sp + L0_OFFSET], %l0
ld [%sp + L1_OFFSET], %l1
ld [%sp + L2_OFFSET], %l2
ld [%sp + L3_OFFSET], %l3
ld [%sp + L4_OFFSET], %l4
ld [%sp + L5_OFFSET], %l5
ld [%sp + L6_OFFSET], %l6
ld [%sp + L7_OFFSET], %l7
ld [%sp + I0_OFFSET], %i0
ld [%sp + I1_OFFSET], %i1
ld [%sp + I2_OFFSET], %i2
ld [%sp + I3_OFFSET], %i3
ld [%sp + I4_OFFSET], %i4
ld [%sp + I5_OFFSET], %i5
ld [%sp + I6_OFFSET], %i6
ld [%sp + I7_OFFSET], %i7
```

(b) Restore local and in from memory

Fig.A14. Code for saving and restoring local and in registers

frame list, which is waitting for saving into the stack in memory, is F. The code segment from line 1 to line 5 is responsible for saving the register file R into current task's TCB, and we achieve assertion marked ②.

The codes from line 6 to 20 saves the prefix F of the frame list into current task's stack in memory. After execution of this segment. The part of the frame list, waitting for storing into memory, becomes empty (nil). And the assertion marked ③ holds.

Then, we prove the code block `switch_new_task`, which restores the context *nst* of the new task $\mathtt{t}_n$. After executing the codes from line 21 to line 28, the context *nst* of the new task $\mathtt{t}_n$ is restored (shown as $\mathsf{Env}((nst, \mathrm{nil}))$), and the assertion marked ④ holds.

Finally, we apply **ABSCSQ** rule, shown in Fig. 19, to execute the abstract assembly primitive switch, and the assertion marked ⑤ holds. By applying **RETL** rule shown in Fig. 19, we finish the proof. □

Theorem 3. $\Psi \vdash C_{\mathtt{switch}} : \{\mathtt{SwitchEntry} \rightsquigarrow \mathsf{switch}\}$.

Proof. We unfold $\Psi \vdash C_{\mathtt{switch}} : \{\mathtt{SwitchEntry} \rightsquigarrow \mathsf{switch}\}$ according to its definition (in Fig. 19) and we need to prove the following hold:

- $\vdash C_{\mathrm{as}} : \Psi$. The correctness proof of this subgoal can be done by apply Lemma 11.
- $\mathsf{wdSpec}(a_{pre}, \mathsf{a}_{post}, \mathsf{switch})$. The correctness proof of this subgoal can be done by apply Lemma 10.

□