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. We have applied it to verify the main body of a context switch routine in a realistic embedded OS kernel. All of the formalization and proofs have been mechanized in Coq.

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. Their correctness 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.* [15, 6, 5]), 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 [1]. 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 does not take effect immediately. Instead the write operation is buffered and then executed X cycles later, where X is a predefined system parameter which usually ranges from 0 to 3.
- *Register windows.* SPARCv8 uses register windows and the window rotation mechanism to avoid saving contexts in the stack directly and achieves high performance in context management.

```
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

We use a simple example in Fig. 1 to show these three features. The function `CALLER` 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 let `npc` points to `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. Its 32 general registers 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 “$\%\mathtt{o}_0 \sim \%\mathtt{o}_7$” for these groups respectively. 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 becomes the in registers in the current window. 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). 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 fully 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. Our work is based on earlier work on assembly code verification but makes the following contributions:

– 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 $\rhd_t \mathtt{sr} \mapsto w$, 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.
– Following SCAP [4], our logic supports modular reasoning of function calls in a direct-style. We use the standard pre- and post-conditions as function specifications, instead of the binary assertion g used in SCAP. This allows us to reuse existing techniques (*e.g.* Coq tactics) to simplify the program verification process. The logic rules for function call and return is 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 [4]) or treats return code pointers as first-class code pointers and gives CPS-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 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.

The program logic, its soundness proof and the verification of the context switch module have been mechanized in Coq[1].

In the rest of paper, we present the program model and operational semantics of SPARCv8 in Sec. 2. Then we propose the program logic in Sec. 3, including the inference rules and the soundness proof. We show the verification of the main body of the context switch module in Sec. 4. Finally we discuss more on related work and conclude in Sec. 5.

[1] We do not give the URL of the code for anonymity. The code is available upon request. The released package does not contain the verified context switch routine, due to copyright concerns.

$$
\begin{array}{rlcl}
\text{(Word)} & w,\,\mathtt{f},\,l & \in & \text{Int32}
\end{array}
$$

$$
\begin{array}{rlcl rlcl}
\text{(Prog)} & \mathrm{P} & ::= & (C, S, \mathtt{pc}, \mathtt{npc}) & \text{(CodeHeap)} & C & \in & \text{Word} \rightharpoonup \text{Comm} \\
\text{(State)} & S & ::= & (M, R, D) & \text{(RState)} & Q & ::= & (R, F) \\
\text{(Memory)} & M & \in & \text{Word} \rightharpoonup \text{Word} & \text{(ProgCount)} & \mathtt{pc},\,\mathtt{npc} & \in & \text{Word} \\
\text{(OpExp)} & \mathtt{o} & ::= & \mathtt{r} \mid w & \text{(AddrExp)} & \mathtt{a} & ::= & \mathtt{o} \mid \mathtt{r} + \mathtt{o}
\end{array}
$$

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

Fig. 2. Machine States and Language for SPARCv8 Code

$$
\begin{array}{rlcl rlcl}
\text{(RegFile)} & R & \in & \text{RegName} \rightharpoonup \text{Word} & \text{(RegName)} & \mathtt{rn} & ::= & \mathtt{r}_0 \mid \ldots \mid \mathtt{r}_{31} \mid \mathtt{psr} \mid \mathtt{sr} \\
\text{(PsrReg)} & \mathtt{psr} & ::= & \mathtt{n} \mid \mathtt{z} \mid \mathtt{v} \mid \mathtt{c} \mid \mathtt{cwp} & \text{(SpeReg)} & \mathtt{sr} & ::= & \mathtt{wim} \mid \mathtt{Y} \mid \mathtt{asr}_0 \mid \ldots \mid \mathtt{asr}_{31} \\
\text{(FrameList)} & F & ::= & \text{nil} \mid \text{fm} :: F & \text{(Frame)} & \text{fm} & := & [w_0, \ldots, w_7] \\
\text{(DelayBuff)} & D & ::= & \text{nil} \mid (t, \mathtt{sr}, w) :: D & \text{(DelayCycle)} & t & \in & \{0, 1, \ldots, X\}
\end{array}
$$

Fig. 3. Register File, Frame List and DelayList

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. 2. The whole program configuration P consists of the code heap C, the machine 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 32-bit integers (called *words*), which can be viewed as memory addresses where the commands are saved. Commands in SPARCv8 can be classified into two categories, the simple instructions `i` and the control-transfer instructions like `call` and `jmp`.

The machine state S consists of three parts: the memory M, the register state Q which is a pair of register file R and frame list F, and the delay buffer D. As defined in Fig. 3, R is a partial mapping from register names to words. Registers include the general registers `r`, the processor state register `psr` and the special registers `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.

Register windows and frame List. SPARCv8 provides 32 general registers, which are split into four 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. The latter three groups (out, local and in) form the current *register window*.

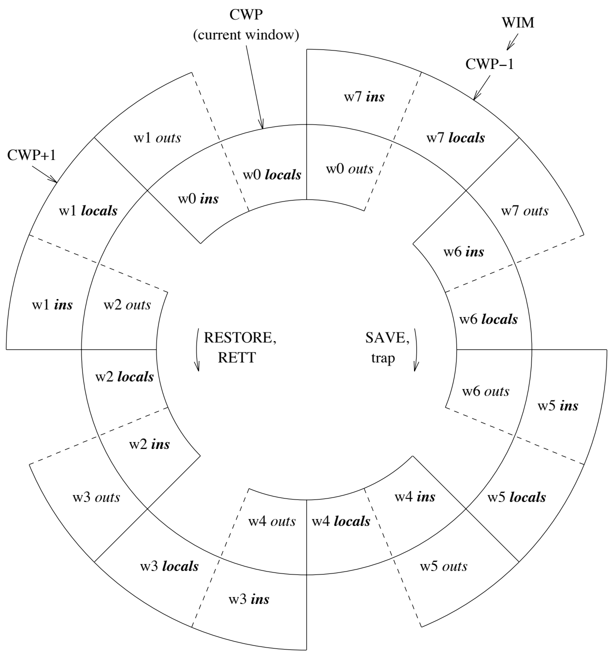

Fig. 4. Register Windows (figure taken from [1])

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. 4, 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 (`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 decrements 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

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

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

$$R\{[\mathtt{r}_i, \ldots, \mathtt{r}_{i+7}] \rightsquigarrow \mathrm{fm}\} \triangleq R\{\mathtt{r}_i \rightsquigarrow w_0\} \ldots \{\mathtt{r}_{i+7} \rightsquigarrow w_7\} \quad \text{where} \quad \mathrm{fm} = [w_0, \ldots, w_7]$$

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

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

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

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

Fig. 5. Auxiliary Definitions for Instruction `save` and `restore`

is valid. We use the assertion $\mathbf{win\_valid}(w_{id}, R)$ defined in Fig. 5 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. 3, 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. 5.

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.

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. 2, we require that a delayed control-transfer instruction 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

We show the selected operational semantics rules in Fig. 6. Before the execution of the instruction pointed by `pc`, the delayed writes in D whose delay cycles are 0 are executed first, as shown in Fig. 6 (a). The execution of the delayed writes are defined in the form of $(R, D) \rightrightarrows (R', D')$, as shown below:

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

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

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

Lemma 2.1. $(R, D) \rightrightarrows (R', D')$ 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, the steps for `save`, `restore` and `wr` instructions, and the steps for other simple instructions, shown in Fig. 6 (b), (c) and (d) respectively.

Note that, after the control-transfer instructions, `pc` is set to `npc` and `npc` contains the target address. 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`). Evaluation of expressions `a` and `o` is defined as $[\![\mathtt{a}]\!]_R$ and $[\![\mathtt{o}]\!]_R$ in Fig. 6 (e).

$$\frac{\begin{array}{l}(R,D) \rightrightarrows (R',D')\\ C \vdash ((M,(R',F),D'),\texttt{pc},\texttt{npc}) \circ\!\!\longrightarrow ((M',(R'',F'),D''),\texttt{pc}',\texttt{npc}')\end{array}}{C \vdash ((M,(R,F),D),\texttt{pc},\texttt{npc}) \longmapsto ((M',(R'',F'),D''),\texttt{pc}',\texttt{npc}')}$$

(a) Program Transistion

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

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

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

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

(b) Control Transfer Instruction Transition

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

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

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

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

(c) Save, Restore and Wr instruction Transition

$$\frac{R(\texttt{sr}) = w \qquad \texttt{r}_d \in \mathrm{dom}(R)}{(M,R) \xrightarrow{\texttt{rd sr r}_d} (M,R\{\texttt{r}_d \rightsquigarrow w\})}$$

$$\frac{R(\texttt{r}_s) = w_1 \quad [\![\texttt{o}]\!]_R = w_2 \quad \texttt{r}_d \in \mathrm{dom}(R)}{(M,R) \xrightarrow{\texttt{add r}_s\ \texttt{o r}_d} (M,R\{\texttt{r}_d \rightsquigarrow w_1+w_2\})}$$

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

(d) Simple Instruction Transition

$$[\![\texttt{o}]\!]_R \triangleq \begin{cases} R(r) & \textbf{if } \texttt{o} = r\\ w & \textbf{if } \texttt{o} = w,\\ & \quad -4096 \le w \le 4095\\ \bot & \textbf{otherwise}\end{cases} \qquad [\![\texttt{a}]\!]_R \triangleq \begin{cases} [\![\texttt{o}]\!]_R & \textbf{if } \texttt{a} = \texttt{o}\\ w_1+w_2 & \textbf{if } \texttt{a} = \texttt{r}+\texttt{o},\ R(\texttt{r})=w_1\\ & \quad \textbf{and } [\![\texttt{o}]\!]_R = w_2\\ \bot & \textbf{otherwise}\end{cases}$$

(e) Expression Semantics

Fig. 6. Selected operational semantics rules

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:

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

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

The `save` and `restore` instruction rotate the register windows and update the register file. Their operations over F and R are defined in Fig. 5.

3 Program Logic

In this section, we introduce the assertion language and program logic designed for SPARCv8 program.

3.1 Assertions

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

Fig. 7. Syntax of Assertions

We define syntax of assertions in Fig. 7, and their semantics in Fig. 8. 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. $l \mapsto w$ specifies a singleton memory cell with value w stored in the address l. $\mathtt{rn} \mapsto w$ says that `rn` is the only register in the register file and it contains the value w. Also `rn` is *not* in the delay list. Separating conjunction $p * q$ has the standard semantics as in separation logic.

The assertion $\rhd_t \mathtt{sr} \mapsto w$ describes a delayed write in the delay buffer D. It describes the uncertainty of `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 `sr` in D (*i.e.* $\mathbf{noDup}(\mathtt{sr}, D)$). This prevents more than one delayed writes to the same register within 4 instruction cycles, which practically have no restrictions on programming. By the semantics we have

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

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:

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

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

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

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

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

Fig. 8. Semantics of Assertions

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

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

Lemma 3.1. $(p * q){\downarrow} \iff (p{\downarrow}) * (q{\downarrow})$.

The lemma can be proved following Lemma 2.1.

We use $\mathtt{cwp} \mapsto (\!| w_{id}, F |\!)$ to describe the pointer $\mathtt{cwp}$ 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. Therefore we have $\mathtt{cwp} \mapsto (\!| w_{id}, F{\cdot}F' |\!) \implies \mathtt{cwp} \mapsto (\!| w_{id}, F |\!)$.

The assertions $\mathsf{a} =_a w$ and $\mathsf{o} = w$ 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 (*i.e.* **word_align**, whose definition is omitted here).

– $\{(\mathrm{fp}, \mathrm{fq})\}$

```
add  %i0, %i1, %l7
add  %l7, %i2, %l7
retl
nop
```

$$\begin{array}{l} \mathrm{fp} \triangleq \lambda\, lv.\, (\%\mathtt{i}_0 \mapsto lv[0]) * (\%\mathtt{i}_1 \mapsto lv[1]) * (\%\mathtt{i}_2 \mapsto lv[2]) \\ \qquad\qquad * \%\mathtt{l}_7 \mapsto \_ * (\mathtt{r}_{15} \mapsto lv[3]) \\ \mathrm{fq} \triangleq \lambda\, lv.\, (\%\mathtt{i}_0 \mapsto lv[0]) * (\%\mathtt{i}_1 \mapsto lv[1]) * (\%\mathtt{i}_2 \mapsto lv[2]) \\ \qquad\qquad * (\%\mathtt{l}_7 \mapsto lv[0] + lv[1] + lv[2]) * (\mathtt{r}_{15} \mapsto lv[3]) \end{array}$$

Fig. 9. Example for Function Specification

3.2 Inference Rules

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

$$\begin{array}{llll} (\mathrm{valList})\ \iota \in \text{list value} & & (\mathrm{pAsrt}) & \mathrm{fp}, \mathrm{fq} \in \mathrm{valList} \rightarrow Asrt \\ (\mathrm{CdSpec})\ \theta ::= (\mathrm{fp}, \mathrm{fq}) & & (\mathrm{CdHpSpec}) & \Psi ::= \{\mathtt{f} \rightsquigarrow \theta\}^* \end{array}$$

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 $lgvl$. They play the role of auxiliary variables — Feeding the pre- and the post-conditions with the same $lgvl$ allows us to establish relationship of states specified in the pre- and post-conditions.

Although we assign a 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 [4], 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.

We give a simple example in Fig. 9 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 $(\mathrm{fp}, \mathrm{fq})$ says that, when provided with the same lv as argument, the function preserves the value of $\%\mathtt{i}_0$, $\%\mathtt{i}_1$ and $\%\mathtt{i}_2$, $\%\mathtt{l}_7$ 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 $(\mathrm{fp}\, lv, \mathrm{fq}\, lv)$ for all lv.

Figure 10 shows selected inference rules in our logic. 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 `i`. The instruction `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 `i` is then used as the precondition to verify the remaining part of the instruction sequence.

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

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

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

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

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

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

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

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

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

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

$$\frac{\begin{array}{c} p \Rightarrow (\texttt{r}_s = w_1 \wedge \texttt{o} = w_2) \qquad w'_{id} = \textbf{next\_cwp}(w_{id}) \qquad w \,\&\, 2^{w'_{id}} = 0 \\ p \Rightarrow (\texttt{cwp} \mapsto \llparenthesis\, w_{id}, F \cdot \_ \cdot \_ \rrparenthesis) * (\mathsf{out} \mapsto \mathrm{fm}_o) * (\mathsf{local} \mapsto \mathrm{fm}_l) * (\mathsf{in} \mapsto \mathrm{fm}_i) * p_1 \\ (\texttt{cwp} \mapsto \llparenthesis\, w'_{id}, \mathrm{fm}_l :: \mathrm{fm}_i :: F \rrparenthesis) * (\mathsf{out} \mapsto \_) * (\mathsf{local} \mapsto \_) * (\mathsf{in} \mapsto \mathrm{fm}_o) * p_1 \Rightarrow \texttt{r}_d \mapsto \_ * p_2 \end{array}}{\vdash \{(\texttt{wim} \mapsto w) * p\}\ \texttt{save}\ \texttt{r}_s\ \texttt{o}\ \texttt{r}_d\ \{(\texttt{wim} \mapsto w) * (\texttt{r}_d \mapsto w_1 + w_2) * p_2\}}\ \textbf{(SAVE)}$$

where $[\texttt{r}_i, \ldots, \texttt{r}_{i+7}] \mapsto [w_0, \ldots, w_7] \ \triangleq\ \texttt{r}_i \mapsto w_0 * \cdots * \texttt{r}_{i+7} \mapsto w_7$
and out, local and in are defined in Fig. 5.

$$\frac{\begin{array}{c} p \Rightarrow (\texttt{r}_s = w_1 \wedge \texttt{o} = w_2) \qquad w'_{id} = \textbf{prev\_cwp}(w_{id}) \qquad w \,\&\, 2^{w'_{id}} = 0 \\ p \Rightarrow (\texttt{cwp} \mapsto \llparenthesis\, w_{id}, \mathrm{fm}_1 :: \mathrm{fm}_2 :: F \rrparenthesis) * (\mathsf{out} \mapsto \_) * (\mathsf{local} \mapsto \_) * (\mathsf{in} \mapsto \mathrm{fm}_i) * p_1 \\ (\texttt{cwp} \mapsto \llparenthesis\, w'_{id}, F \cdot \_ \cdot \_ \rrparenthesis) * (\mathsf{out} \mapsto \mathrm{fm}_i) * (\mathsf{local} \mapsto \mathrm{fm}_1) * (\mathsf{in} \mapsto \mathrm{fm}_2) * p_1 \Rightarrow \texttt{r}_d \mapsto \_ * p_2 \end{array}}{\vdash \{(\texttt{wim} \mapsto w) * p\}\ \texttt{restore}\ \texttt{r}_s\ \texttt{o}\ \texttt{r}_d\ \{(\texttt{wim} \mapsto w) * (\texttt{r}_d \mapsto w_1 + w_2) * p_2\}}\ \textbf{(RESTORE)}$$

Fig. 10. Seleted Inference Rules

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

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. 8). At the end of the delayed write, we use $\triangleright_3 \mathtt{sr} \mapsto w_1 \oplus w_2$ 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. 5 and 6.

3.3 Semantics and Soundness

We first define the safety of instruction sequences, $\mathsf{safe\_insSeq}(C, S, \mathtt{pc}, \mathtt{npc}, q, \Psi)$. It says C can execute safely from S, `pc` and `npc` until reaching the end of the current instruction sequence ($C[\mathtt{pc}]$), and q holds if $C[\mathtt{pc}]$ ends with the return instruction. It is formally defined in Def. 3.2. Here we use "$\_ \longmapsto^n \_$" to represent n-step execution.

Definition 3.2 (Safety of Instruction Sequences).
$\mathsf{safe\_insSeq}(C, S, \texttt{pc}, \texttt{npc}, q, \Psi)$ holds if and only if the following are true (we omit the case for `be` here, which is similar to `jmp`):

- if $C(\texttt{pc}) = \texttt{i}$ then :
  - there exist $S', \texttt{pc}, \texttt{npc}'$, such that $C \vdash (S, \texttt{pc}, \texttt{npc}) \longmapsto (S', \texttt{pc}', \texttt{npc}')$,
  - for any $S', \texttt{pc}', \texttt{npc}'$, if $C \vdash (S, \texttt{pc}, \texttt{npc}) \longmapsto (S', \texttt{pc}', \texttt{npc}')$, then $\mathsf{safe\_insSeq}\ (C, S', \texttt{pc}', \texttt{npc}', q, \Psi)$
- if $C(\texttt{pc}) = \texttt{jmp a}$ then :
  - there exist $S', \texttt{pc}', \texttt{npc}'$, such that $C \vdash (S, \texttt{pc}, \texttt{npc}) \longmapsto^2 (S', \texttt{pc}', \texttt{npc}')$,
  - for any $S', \texttt{pc}', \texttt{npc}'$, if $C \vdash (S, \texttt{pc}, \texttt{npc}) \longmapsto^2 (S', \texttt{pc}', \texttt{npc}')$, then there exist $\mathrm{fp}, \mathrm{fq}, \iota$ and p_r, such that the following hold:
    (1) $\texttt{npc}' = \texttt{pc}'{+}4$, $\Psi(\texttt{pc}') = (\mathrm{fp}, \mathrm{fq})$,
    (2) $S' \models (\mathrm{fp}\ \iota) * p_r$, $(\mathrm{fq}\ \iota) * p_r \Rightarrow q$.
- if $C(\texttt{pc}) = \texttt{be f}$ then ...
- if $C(\texttt{pc}) = \texttt{call f}$ then :
  - there exist $S', \texttt{pc}', \texttt{npc}'$, such that $C \vdash (S, \texttt{pc}, \texttt{npc}) \longmapsto^2 (S', \texttt{pc}', \texttt{npc}')$,
  - for any $S', \texttt{pc}'$ and $\texttt{npc}'$, if $C \vdash (S, \texttt{pc}, \texttt{npc}) \longmapsto^2 (S', \texttt{pc}', \texttt{npc}')$, then there exist $\mathrm{fp}, \mathrm{fq}, \iota$ and p_r, such that the following hold:
    (1) $\texttt{npc}' = \texttt{pc}'{+}4$, $\Psi(\texttt{pc}') = (\mathrm{fp}, \mathrm{fq})$,
    (2) $S' \models (\mathrm{fp}\ \iota) * p_r$,
    (3) for any S', if $S' \models (\mathrm{fq}\ \iota) * p_r$, then $\mathsf{safe\_insSeq}\ (C, S', \texttt{pc}+8, \texttt{pc}+12, q, \Psi)$,
    (4) for any S', if $S' \models (\mathrm{fq}\ \iota)$, then $S'.Q.R(\texttt{r}_{15}) = \texttt{pc}$.
- if $C(\texttt{pc}) = \texttt{retl}$ then :
  - there exist $S', \texttt{pc}', \texttt{npc}'$, such that $C \vdash (S, \texttt{pc}, \texttt{npc}) \longmapsto^2 (S', \texttt{pc}', \texttt{npc}')$,
  - for any $S', \texttt{pc}'$ and $\texttt{npc}'$, if $C \vdash (S, \texttt{pc}, \texttt{npc}) \longmapsto^2 (S', \texttt{pc}', \texttt{npc}')$, then $S' \models q$, $\texttt{pc}' = S'.Q.R(\texttt{r}_{15}){+}8$, and $\texttt{npc}' = S'.Q.R(\texttt{r}_{15}){+}12$.

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

Definition 3.3 (Judgment Semantics).

- $\Psi \models \{(p, q)\}\ \texttt{f} : \mathbb{I}$ if and only if, for all C and S such that $C[\texttt{f}] = \mathbb{I}$ and $S \models p$, we have $\mathsf{safe\_insSeq}(C, S, \texttt{f}, \texttt{f}{+}4, q, \Psi)$.
- $\models C{:}\Psi$ if and only if, for all $\texttt{f}$, fp and fq such that $\Psi(\texttt{f}) = (\mathrm{fp}, \mathrm{fq})$, we have $\Psi \models \{(\mathrm{fp}\ \iota, \mathrm{fq}\ \iota)\}\ \texttt{f} : C[\texttt{f}]$ for all ι.

Next we define the safety $\mathsf{safe}^n(C, S, \texttt{pc}, \texttt{npc}, q, k)$ of whole program execution. It says that, starting with `pc`, `npc` and the state S, and with the depth k of function calls, the code C either *halts* in less than n steps, with the final state satisfies q, or it executes at least n steps safely. Here we say C halts if it reaches the return point of the topmost function (when the depth k of the function call is 0). In the definition below, the depth k increases by the `call` instruction and decreases by `retl` (unless $k = 0$).

Definition 3.4 (Program Safety). $\mathsf{safe}^0(C, S, \texttt{pc}, \texttt{npc}, q, k)$ always holds.
$\mathsf{safe}^{n+1}(C, S, \texttt{pc}, \texttt{npc}, q, k)$ holds if and only if the following are true:

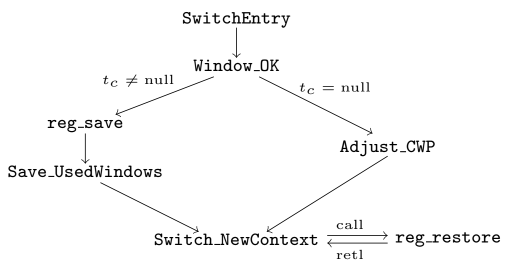

Fig. 11. The Structure of Context Switch Module

1. if $C(\texttt{pc}) \in \{\texttt{i}, \texttt{jmpl a}, \texttt{be f}\}$, then:
   - there exist $S', \texttt{pc}', \texttt{npc}'$, such that $C \vdash (S, \texttt{pc}, \texttt{npc}) \longmapsto (S', \texttt{pc}', \texttt{npc}')$;
   - for any $S', \texttt{pc}', \texttt{npc}'$, if
     $C \vdash (S, \texttt{pc}, \texttt{npc}) \longmapsto (S', \texttt{pc}', \texttt{npc}')$, then $\mathsf{safe}^n\,(C, S', \texttt{pc}', \texttt{npc}', q, k)$;
2. if $C(\texttt{pc}) = \texttt{call f}$, then:
   - there exist $S', \texttt{pc}', \texttt{npc}'$ such that $C \vdash (S, \texttt{pc}, \texttt{npc}) \longmapsto^2 (S', \texttt{pc}', \texttt{npc}')$;
   - for any $S', \texttt{pc}', \texttt{npc}'$, if $C \vdash (S, \texttt{pc}, \texttt{npc}) \longmapsto^2 (S', \texttt{pc}', \texttt{npc}')$,
     then $\mathsf{safe}^n(C, S', \texttt{pc}', \texttt{npc}', q, k+1)$;
3. if $C(\texttt{pc}) = \texttt{retl}$, then:
   - there exist $S', \texttt{pc}', \texttt{npc}'$ such that $C \vdash (S, \texttt{pc}, \texttt{npc}) \longmapsto^2 (S', \texttt{pc}', \texttt{npc}')$;
   - for any $S', \texttt{pc}', \texttt{npc}'$, if $C \vdash (S, \texttt{pc}, \texttt{npc}) \longmapsto^2 (S', \texttt{pc}', \texttt{npc}')$, then
     if $k = 0$ then
     $\quad S' \models q$
     else
     $\quad \mathsf{safe}^n(C, S', \texttt{pc}', \texttt{npc}', q, k-1)$.

Then the following theorem and corollary show the soundness of our logic.

Theorem 3.5 (Soundness). $\vdash C : \Psi \Longrightarrow \models C : \Psi$

Corollary 3.6 (Function Safety). If $\Psi \models \{(p, q)\}\,\texttt{pc} : C[\texttt{pc}]$, $S \models p$, and $\models C : \Psi$, then $\forall n.\ \mathsf{safe}^n(C, S, \texttt{pc}, \texttt{pc}{+}4, q, 0)$.

4 Verifying a Realistic Context Switch Module

We apply our program logic to verify the main body of a context switch routine implemented in SPARCv8, which is used to save the current task's context and restore the new task's context. Figure 11 shows the structure of the code.

- `SwitchEntry` is the entry of the module. It checks `SwitchFlag` to see if a context switch is needed. If yes, it enters the `Window_OK` block.
- `Window_OK` checks if the current task is null (which may happen if the switch follows the delete of the current task). If yes, it jumps to `Adjust_CWP`, which resets the pointer `cwp` of the current register window so that it points to the last valid window. It essentially pops all the frames to empty the circular stack of register windows. If the current task is *not* null, it calls `reg_save` to save the general registers into the TCB, and then enter the code block `Save_UsedWindows` to save other register windows (F in our state model).

– `Save_UsedWindows` saves the register windows (except the current one) into the current task's stack in memory.
– `Switch_NewContext` restores the general registers and other register windows from the new task's TCB 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 manages the register windows. To save all the 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 5 arguments, the id of the current task t_c, the id of the new task t_n, the value *flag* of the SwitchFlag, the values *env* of general registers and all other register windows, and the new task's context *nst* that needs to be restored.

$$
\begin{aligned}
\mathsf{a}_{pre}(t_c, t_n, \mathit{flag}, \mathit{env}, \mathit{nst}) &\triangleq \mathsf{Env}(\mathit{env}) * (\mathsf{SwitchFlag} \mapsto \mathit{flag}) * (\mathsf{TaskNew} \mapsto t_n) * \\
&\qquad (\mathit{flag} = \text{false} \vee \mathsf{CurT}(t_c, \_, \mathit{env}) * \mathsf{NoCurT}(t_n, \mathit{nst})) \\
\mathsf{a}_{post}(t_c, t_n, \mathit{flag}, \mathit{env}, \mathit{nst}) &\triangleq \exists \mathit{env}'.\ \mathsf{Env}(\mathit{env}') * (\mathsf{SwitchFlag} \mapsto \text{false}) * (\mathsf{TaskNew} \mapsto t_n) * \\
&\qquad (\mathit{flag} = \text{false} \wedge \mathsf{p\_env}(\mathit{env}) = \mathsf{p\_env}(\mathit{env}') \\
&\qquad\quad \vee (\mathsf{CurT}(t_n, \mathit{nst}, \mathit{env}') \wedge \mathsf{p\_env}(\mathit{env}') = \mathit{nst}) * \\
&\qquad\qquad \mathsf{NoCurT}(t_c, \mathsf{p\_env}(\mathit{env})))
\end{aligned}
$$

In the specification, we use $\mathsf{Env}(\mathit{env})$ to specify the values of general registers and the register windows. The variable TaskNew records the identifier of the new task. If SwitchFlag is false, we do not need any knowledge about the current and the new tasks since there is no context switch. Otherwise we describe the state of the current task (its TCB and stack in memory) using $\mathsf{CurT}(t_c, \_, \mathit{env})$, and the saved context of the new task using $\mathsf{NoCurT}(t_n, \mathit{nst})$. Due to space limitation we omit the detailed definitions here.

If we compare a_{pre} and a_{post}, we can see that t_n becomes the current task ($\mathsf{CurT}(t_n, \mathit{nst}, \mathit{env}')$), and its general registers and stack, specified by $\mathsf{Env}(\mathit{env}')$, are loaded from the saved context *nst* (*i.e.* $\mathsf{p\_env}(\mathit{env}') = \mathit{nst}$). Here $\mathsf{p\_env}(\mathit{env}')$ refers to the part of the environment that we want to save or restore as context. Correspondingly, t_c becomes non-current-thread, and part of its environment *env* at the entry of the context switch is saved, as specified by $\mathsf{NoCurT}(t_c, \mathsf{p\_env}(\mathit{env}))$.

We omit the code manages interrupt and float registers in the original system, which are not supported in our logic. The segment we verify has around 250 lines of assembly code, and we verify it by 6690 lines of Coq proof scripts.

5 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 [10, 2, 7, 8, 16, 11, 13]. SCAP [4] 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. Also SCAP is based on a simplified 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 [9] 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 [2], Tan and Apple present a program logic $\mathcal{L}_c$ for reasoning about control flow in assembly code [13]. Although $\mathcal{L}_c$ is implemented on top of SPARC machine language, the underlying logic is a type system instead of a full-blown program logic for functional correctness. Handling SPARC features such as delayed writes or delayed control transfers is not the focus of $\mathcal{L}_c$. They also reason about functions in the the continuation-passing style. Wang *et al.* [14] formalize the SPARCv8 ISA in Coq but give no program logic. Our operational semantics of SPARCv8 follows their work.

Ni *et al.* [12] verify a context switch module of 19 lines in x86 code to show case the support of embedded code pointers (ECP) in XCAP [11]. The context switch module we verify comes from a practical OS kernel, which is more realistic and consists of more than 250 lines of assembly code, but our logic does not really support the switch of return addresses, which requires further extension like [3]. Our focus is to verify the code manages the register windows, and the function calls made internally.

Conclusion. We present a program logic for SPARCv8. Our logic is based on a realistic semantics model and supports main features of SPARCv8, including delayed control transfer, delayed writes, and register windows. We have applied the program logic to verify the main body of the context switch routine in a realistic embedded OS kernel. Our current work can only handle sequential SPARCv8 program verification for partial correctness. We will extend it for concurrency and refinement verification in the future. Also we would like to link the verified inline assembly with verified C code for whole system verification.

References

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

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

[3] X. Feng, Z. Ni, Z. Shao, and Y. Guo. An open framework for foundational proof-carrying code. In *TLDI*, pages 67–78, 2007.

[4] X. Feng, Z. Shao, A. Vaynberg, S. Xiang, and Z. Ni. Modular Verification of Assembly Code with Stack-Based Control Abstractions. In *PLDI*, June 2006.

[5] R. Gu, J. Koenig, T. Ramananandro, Z. Shao, X. N. Wu, S.-C. Weng, H. Zhang, and Y. Guo. Deep specifications and certified abstraction layers. In *POPL*, pages 595–608, Jan 2015.

[6] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal Verification of an OS Kernel. In *SOSP*, pages 207–220, Oct 2009.

[7] G. Morrisett, K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich, and S. Zdancewic. Talx86:a realistic typed assembly language. In *1999 ACM SIGPLAN Workshop on Compiler Support for System Software*, pages 25–35, May 1996.

[8] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. In *POPL*, pages 85–97, Jan 1998.

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

[10] G. C. Necula and P. Lee. Safe Kernel Extensions Without Run-Time Checking. In *Proc.2nd USENIX Symp. on Operating System Design and Impl*, pages 229–243, 1996.

[11] Z. Ni and Z. Shao. Certified Assembly Programming with Embedded Code Pointers. In *POPL*, pages 320–333, 2006.

[12] Z. Ni, D. Yu, and Z. Shao. Using XCAP to Certify Realistic Systems code: Machine context management. In *TPHOLs*, Sept 2007.

[13] G. Tan and A. W. Appel. A compositional logic for control flow. In *VMCAI*, Jan 2006.

[14] J. Wang, M. Fu, L. Qiao, and X. Feng. Formalizing SPARCv8 Instruction Set Architecture in Coq. In *SETTA*, Oct 2017.

[15] F. Xu, M. Fu, X. Feng, X. Zhang, H. Zhang, and Z. Li. A practical verification framework for preemptive os kernels. In *CAV*, pages 59–79, July 2016.

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