Noname manuscript No.
(will be inserted by the editor)

An Isabelle/HOL formalisation of the SPARC instruction set architecture and the TSO memory model

Zhé Hóu · David Sanan · Alwen Tiu · Yang Liu · Koh Chuen Hoa · Jin Song Dong

Accepted to publish in the Journal of Automated Reasoning on 6/8/2020.

Keywords Instruction Set Architecture; Form Verification; Isabelle/HOL; Weak Memory Model; TSO

Abstract The SPARC instruction set architecture (ISA) has been used in various processors in workstations, embedded systems, and in mission-critical industries such as aviation and space engineering. Hence, it is important to provide formal frameworks that facilitate the verification of hardware and software that run on or interface with these processors. In this work, we give the first formal model for *multi-core* SPARC ISA and Total Store Ordering (TSO) memory model in Isabelle/HOL.

We present two levels of modelling for the ISA: The low-level ISA model, which is *executable*, covers many features specific to SPARC processors, such as delayed-write for control registers, windowed general registers, and more complex memory access. We have tested our model extensively against a LEON3 simulation board, the test covers both single-step executions and sequential execution of programs. We also prove some important properties for our formal model, including a non-interference property for the LEON3 processor.

The high-level ISA model is an abstraction of the low-level model and it provides an interface for memory operations in multi-core processors. On top of the

Z. Hóu (Corresponding Author)
Institute for Integrated and Intelligent Systems, Griffith University, Australia
E-mail: z.hou@griffith.edu.au

D. Sanan
School of Computer Science and Engineering, Nanyang Technological University, Singapore

Alwen Tiu
College of Engineering and Computer Science, The Australian National University, Australia

Yang Liu
School of Computer Science and Engineering, Nanyang Technological University, Singapore

Koh Chuen Hoa
Singapore Defence Science Organisation, Singapore

Jin Song Dong
School of Computing, National University of Singapore

high-level ISA model, we formalise two TSO memory models: one is an adaptation of the axiomatic SPARC TSO model [49,51], the other is a new operational TSO model which is suitable for verifying execution results. We prove that the operational model is sound and complete with respect to the axiomatic model. Finally, we give verification examples with two case studies drawn from the SPARCv9 manual.

1 Introduction

Formal models of instruction set architectures (ISAs) not only provide a rigorous understanding of the semantics for instructions, but also are useful in verifying low-level programs such as hardware drivers, virtual machines, compilers, etc. Defining an ISA model in a theorem prover opens up the possibility to reason about properties and semantics of the ISA and machine code. For an extensively developed application of an ARMv7 formal model, see Khakpour et al.'s work on verifying non-interference at the ISA level [29]. There have been various publicly available formal models for ISAs in the literature, e.g., for ARM6 [17], ARMv7 [20], x86 [46]. However, to the best of our knowledge, there are no formalisations of SPARC *multi-core* processors.

The SPARC architecture has many important applications. For instance, SPARC was commonly used in Sun Oracle station in 2010 when it was acquired by Oracle. Oracle then launched many SPARC based servers, such as Sun Blade Servers and Sun Netra Carried-Grade Servers [14]. SPARC is also used in supercomputers. Fujitsu's K computer [21], ranked NO.1 in TOP500 2011, combined 88,128 SPARC CPUs. Tianhe-2 [3], ranked NO.1 in TOP500 2014, has a number of components with SPARC based processors. Most importantly, SPARC is widely used in defense, aviation systems, and space missions. The European Space Agency (ESA) chose to use SPARCv8, mainly because SPARC is one of the few fully open ISAs (other than RISC-V [2] etc.), and has significant support. The ESA then started the LEON project to develop processors for space projects [15].

Background project. This work is a part of a research project called Securify, which aims to verify an execution stack ranging from CPU, micro-kernel, libraries to applications. We use a multi-layer verification approach where we formalise each layer separately and use a refinement-based approach to show that properties proved at the top level are preserved at the lower levels. One such property is a non-interference property between different partitions in a micro-kernel. We have recently completed a formalisation of the high-level specification of a separation micro-kernel [55], and the idea is to show that the implementation of such a micro-kernel preserves the *non-interference* property, both at the software and the hardware level. As a concrete case study, we choose to formalise the XtratuM [53] micro-kernel that runs on top of the multi-core LEON3 processor; these formalisation efforts are still on-going. The ISA formalisation described in this paper is a key component bridging these two formalisations. Our choice of XtratuM and LEON3 is mainly driven by the fact that they are open source and that our intended applications will be built on these platforms. Our model can be instantiated to LEON2 and LEON4, we do not use the latter because its source code is not available. Since our goal is to support the verification

of XtratuM machine code, we currently focus on formalising the *integer unit (IU)* of SPARCv8, which contains all the instructions used in XtratuM. Also derived from the project requirement is the focus on the *total store ordering* (TSO) memory model for multi-core applications, because the XtratuM hypervisor follows the TSO standard.

Low-level ISA model. We present a low-level Isabelle/HOL model for the IU of the SPARCv8 ISA. Although there are formal models for other ISAs in the literature (e.g., [17,46]), the difference in architecture and several special features of SPARC make the adaptation of existing models to our work challenging. For example, the register model in SPARCv8 is not a flat 32-register model, but instead consists of a set of overlapping register windows arranged in a circular buffer. There are flags such as *annul* that may cause instructions to be skipped [14]. Memory access in SPARCv8 requires an additional parameter, i.e., the address space identifier (ASI), that specifies whether the processor is in supervisor or user mode, and whether the memory access is data access. Finally, the write control register instructions may be delayed, thus we have to devise a mechanism to perform delayed executions. A similar feature appears in the MIPS architecture, which is modeled in L3 [1]. Our model covers the following aspects of IU: control registers, system registers, and general registers; operations on registers (e.g., RDPSR, WRPSR, etc.); a strong consistency memory model with treatments for address spaces; a simple cache model with write-through policy; flags such as *annul*, signals such as $execute\_mode$ and $error\_mode$; and a trap (exception and interruption) model with all the trap table assignments. We also model store barrier and flush. Except for hardware signals and interrupts, we have captured all the details of the IU defined in Appendix C of the SPARCv8 manual [51]. We also provide a memory management unit (MMU) model to support multi-core microkernel verification. Although our model does not cover the co-processor unit and the float-point unit, they can be added to our model using the same methodology.

High-level ISA model and TSO model. The low-level ISA model was published at FM 2016 [28]. This journal paper extends the FM 2016 paper with two components: a high-level ISA model, which serves as the interface for memory operations, and two TSO memory models. The high-level ISA model simplifies the low-level ISA model by focusing on a subset of the operational semantics that are used in multi-core processors and memory operations. The semantics of the high-level model considers the execution of a "block" of instructions at a time, each block corresponds to a memory operation. The semantics for executing each individual instruction remains the same as that in the low-level model, thus the soundness at the instruction level is retained. On top of the abstract ISA model, we give two TSO models: the first one is a formalisation of the axiomatic SPARC TSO model [49,51]; the second one is an operational TSO model. The axiomatic SPARC TSO model stands as a reference for the correctness of the memory model. It can be used to verify the memory operation order with respect to the program order, but it is difficult to use the axioms to reason about program executions step by step. For this purpose, we develop the operational TSO model to reason about program executions. The integration of instruction semantics and weak memory model is essential to support formal reasoning about concurrent programs, but this problem is sometimes neglected in the weak memory

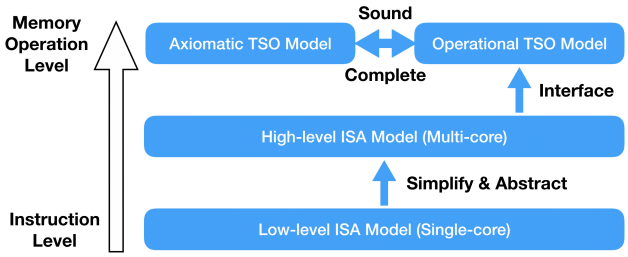

Fig. 1: An overview of the formalisations in this paper.

literature [48]. We show that the operational TSO model is sound and complete with respect to the axiomatic model. That is, every execution given by the operational model conforms with the axioms, and every sequence of memory operations that conforms with the axioms can be executed by the operational model. Finally, we give two case studies based on the "Indirection Through Processors" program and spin lock with CASA, both of which are drawn from the SPARC manual, to exemplify verifications on the order of memory operations as well as on the result of execution.

An overview of this work is shown in Figure 1.

Contributions. We summarise the main contributions as follows, where we indicate the parts that were published in FM 2016 by references:

1. We give a low-level ISA model for the integer unit of SPARCv8 ISA [28].
2. The low-level model can be exported to OCaml code for both single step execution and sequential execution [28].
3. The instruction semantics has been extensively tested against a LEON3 simulation board [28].
4. We show a non-interference property for the LEON3 processor [28].
5. We give a high-level ISA model to interface with the memory model *(new)*.
6. We give two TSO models: an axiomatic model and an operational model *(new)*.
7. We show that the operational TSO model is sound and complete w.r.t the axiomatic TSO model *(new)*.
8. We give two verification case studies of multi-core programs *(new)*.

Instruction coverage summary. The SPARCv8 ISA contains a total of 202 instructions for the integer unit, the floating-point unit and the co-processor unit. Our model formalises all the 116 instructions for the integer unit, including the instructions for branching, memory load and store, arithmetic, logic, shifting, control transfer, traps, etc. The high-level model additionally formalises the atomic compare and swap (CASA) instruction from the SPARCv9 architecture; this instruction is implemented in some SPARCv8 processors of our interest.

The complete source code of our formalisation, proofs, and exported executable code can be found at the Securify project website [47].

Related work. Santoro et al. [45] gave an executable specification for the SPARCv9 architecture with Rapide. However, their model is not built in a theorem prover; thus it is not suitable for formal verification purposes. Lim and Reps [32] developed the TSL framework for analysing machine code. Their framework is generic and can be used to model different ISAs such as X86 and PowerPC. Interestingly, the authors have also developed model checkers based on TLS that can perform program analysis tasks. Our work shares the common goal of formal verification for low-level code, but we are also interested in proving architecture-wide properties such as safety and security. Thus, we chose to use theorem proving instead of model checking.

Roşu and Şerbănuţă developed the K framework [42], which can be used to define programming language semantics, formal analysis tools, among others. The K framework is built towards the direction of a unified theory of operational and axiomatic semantics [43], and it is used to model very comprehensive X86-64 ISA semantics [13]. This framework provides an impressive suite of features that facilitate formal program analysis and verification. Again, besides verifying the correctness of programs, we are also interested in other properties that we found easier to formalise in a theorem prover such as Isabelle/HOL.

Fox studied verification of the ARM6 micro-architecture at the RTL level [17]. Fox and Myreen later gave more detailed models for ARM ISAs ranging from version 4 to version 7. Their model for ARMv7 uses monadic specifications and covers details from instruction decoding to operational semantics in the architecture [20]. Their ARMv7 model is the closest work to ours, and it provides a good methodological direction for formalising an ISA and validating the model. Fox et al. then started a project to specify various ISAs using a specification language called L3 [18, 1]. Fox recently developed a framework for formal verification of ISAs [19]. The framework consists of the L3 language for modelling ISAs, Standard ML for efficient emulation, and HOL4 for formal reasoning. On validation, we mainly test our model using randomly generated instructions. This is a standard method used in [20] and [10]. There are also formal models for the x86 architecture, such as Sarkar et al.'s work on the semantics of x86-CC machine code [46]. Another interesting work is the ACL2 ISA models [24]. Similarly to our work, the ACL2 ISA models define instruction semantic functions over states and provide functions for executing the model for one instruction or sequentially. Also, ACL2's abstract `stobjs` can be used to define the state of an x86 ISA model [23]. A difference is that the ACL2 models are more general, whereas our model is more specific and detailed for SPARCv8. The advantage of using ACL2 is that ACL2 naturally supports fast evaluation. The Compcert project gave a formally verified compiler for PowerPC, ARM, and IA32 processors [30, 31]. A remotely related work is Liu and Moore's executable JVM model M6 [33], which is written in a subset of Common Lisp and allows for analytical reasoning as well as simulation. Finally, the JVM specification given by Atkey [7] inspired us to define the model in a proof assistant which supports code export for execution.

There is a rich literature on relaxed memory models, but most of them do not consider machine code semantics. Here we only discuss the most closely related ones. Typically memory models appear in two forms: axiomatic model and operational model. The axiomatic TSO memory model for SPARC is given by Sindhu and Frailong [49]. This model is used in the SPARCv8 manual [51] and is later referred to

as the "golden memory model" [34]. Petri and Boudol [40,8] give a comprehensive study on various weak memory models, including SPARC TSO, PSO, and RMO. They show that the store buffer semantics of TSO and PSO corresponds to their semantics of "speculations". Gray and Flur et al. [25,16] have established axiomatic and operational models for TSO and their equivalence. Their work is also integrated with detailed instruction semantics for x86, IBM Power, ARM, MIPS, and RISC-V. They have developed a language called Sail for expressing sequential ISA descriptions with relaxed memory models that later can be translated into Isabelle/HOL. However, the current set of modelled ISA does not include any variance for the SPARC ISA. Although it would have been possible to rewrite the semantics of [28] in Sail, this language lacks some important features necessary for our work. First, Sail does not provide some low-level system semantics such as exceptions and interrupts; second, their framework does not include an execution model for multi-core processors. Higham et al. and Burckhardt et al. [9,27] have surveyed numerous weak memory models, including SPARC TSO and PSO in both axiomatic and operational styles. The latter work also presents algorithms for verifying store buffer safety.

Besides Burckhardt's work, there are other tools and techniques developed for verifying memory operations. Notably, Hangal et al.'s TSOtool [26] is a program for checking the behaviour of the memory subsystem in a shared memory multiprocessor computer aginst the TSO specification. Although verifying TSO compliance is an NP-complete problem, the authors give a polynomial time incomplete algorithm to efficiently check memory errors. Companies such as Intel also actively work on tools for efficient memory consistency verification [44]. Roy et al.'s tool is also polynomial time and is deployed across multiple groups at Intel. A tool specialised for SPARC instructions is developed by Park and Dill [39]. Recently, Lusting et al. introduced PipeCheck [35], a tool developed in the Coq theorem prover for the specification and verification of memory consistency models at the architectural level. This tool is focused on checking memory consistency along the different pipeline stages of the micro-architecture rather than providing a model useful from the programmer's point of view. Alglave et al. [5] developed a simulation tool named herd that allows the user to specify a variety of weak memory models for different architectures. Much like our work, they also provide axiomatic and operational semantics for memory models and show their equivalence. On the other hand, their work is more general and can be instantiated for Power, ARM and X86, whereas our work is focused on unique and detailed features of SPARC. Another closely related work, although not mechanised in a theorem prover, is Pulte et al.'s relaxed memory models for ARM [41].

There are also memory models that are formalised in theorem provers, such as Yang et al.'s axiomatic Itanium model Nemos in SAT solvers and Prolog [54] and the Java Memory Model in Isabelle/HOL [6]. Alglave et al. formalised a class of axiomatic relaxed memory models in Coq [4]. Crary and Sullivan formalised a calculus in Coq for relaxed memory models [12]. Their calculus is more relaxed than existing architectures, and their work is intended to serve as a programming language. A more related work is Owens, Sarkar, Sewell, et al.'s formalisation of x86 ISA and memory models [46,38,48]. They formalise both the ISA and related memory models such as x86-CC and x86-TSO in HOL and show the correspondence between different styles of memory models. Whilst the X86-TSO programmer's model presented

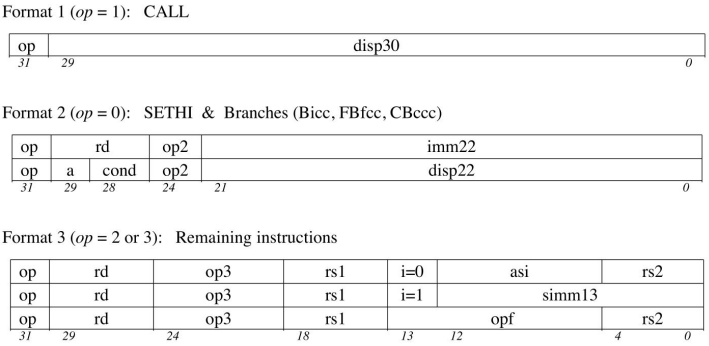

Fig. 2: The formats for SPARCv8 instructions. Source: [51].

in [48] simplifies the hardware memory axioms for easier reasoning, it still keeps low-level structures such as the FIFO store buffers that we remove in our operational semantics. In this sense, reasoning about our memory model is even simpler than in the memory abstraction introduced in [48]. It is possible to translate Gray and Flur et al.'s work [25, 16] to Isabelle/HOL or Coq code. However, the resulting formal model would rely on the correctness of the translation tool such as Lem [36], which adds one more layer of complication for verification.

2 Background

This section introduces the necessary background of the SPARCv8 architecture and the monadic modeling approach.

2.1 Overview of SPARCv8 ISA

The IU of SPARCv8 contains 40 to 520 general-purpose registers depending on the implementation. The IU also controls the overall operation of the processor, thus it is a major part of the processor. All SPARCv8 instructions are 32-bit wide. Instructions in the IU fall into four categories: (1) load/store; (2) arithmetic/logical/shift; (3) control transfer; (4) read/write control register. There are only three instruction formats, shown in Fig. 2. The load and store instructions are the only instructions that access memory. SPARC only has two addressing modes: a memory address is given by either two registers or a register and a signed 13-bit immediate value. Most instructions operate on two registers, and write the result in the third register. Traps are vectorised through a table, and cause an allocation of a fresh register window in the register file. The main special features of SPARCv8 are highlighted below.

Windowed registers. Unlike other architectures, the general purpose registers in the SPARC architecture are grouped in overlapping windows. This design allows for straightforward, high-performance compilers and a significant reduction in memory

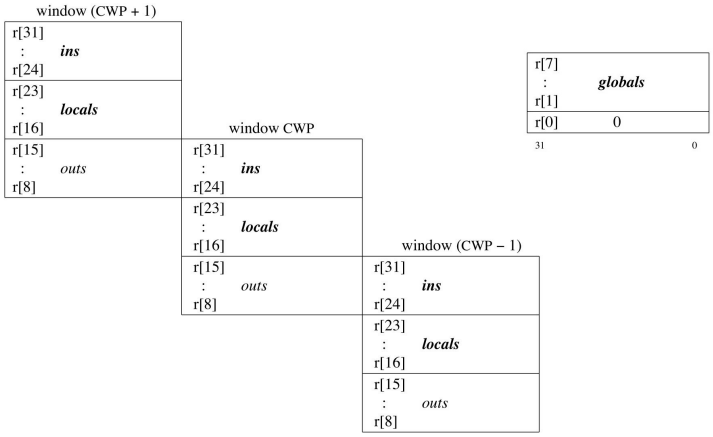

Fig. 3: Three overlapping windowed registers and the global registers. Source: [51].

load/store instructions over other RISCs [51]. A *window* contains 8 *in* registers, 8 *local* registers, and 8 *out* registers. At a given time, an instruction can access 8 *global* registers and the 24 registers in the current window. The *in* registers of the current window are the *out* registers of the next window; the *out* registers of the current window are the *in* registers of the previous window; see Fig. 3. The windows are arranged in a circular buffer, where the last window's *out registers* overlap with the first window's *in* registers. The current window is determined by a segment in the processor state register (PSR). The Window Invalid Mask (WIM) register keeps a bit map that contains information about which windows are currently invalid.

Address space identifier. The memory model in SPARCv8 contains a linear 32-bit address space. When the IU accesses memory, it appends to the address an address space identifier (ASI), which encodes whether the processor is in supervisor or user mode and whether the access is to instruction memory or to data memory, among others. The ASI is also used to access device registers and perform certain operations on devices. The SPARC architecture defines 4 of the 256 address spaces: user instruction, user data, supervisor instruction, and supervisor data [51].

Delayed-write. Besides the general registers, there are also control registers such as the PSR. The write instructions for control registers are delayed-write instructions. 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" [51].

Signals. There are some signals either from instructions or from hardware that play important roles in the execution of instructions. For example, SPARC, like other RISC ISAs, features delayed control transfer instructions. When a delayed (conditional) jump instruction is executed, the jump is not effected immediately. Rather, the

next instruction (also referred to as the delay slot) will be executed before the control transfer to the jump location is done. However, the delayed control transfer instructions in SPARC may contain an annul bit that signals that the instruction in the delay slot is to be skipped. We thus need to keep track of such information in the state and use it to determine whether certain instructions are to be skipped or not.

2.2 Monads in Operational Semantics

As with the ARMv7 formalisation [20], we use sequential monads to define operations in the ISA. A monad is an abstract data type that represents computations. Our Isabelle monad library is a modified version of the one used in NICTA's seL4 project [11]. Instead of using non-deterministic monads in [11], here we use deterministic monads (cf. Section 3.2 for reasons) defined as below, where M is a shorthand for det_monad.

$$\textbf{type\_synonym}\ ('s,'a)\ M = \text{“}'s \Rightarrow ('a \times 's) \times bool\text{”}$$

which returns a pair $('a \times 's)$ of the result and the next state, and also a failure flag. A 'true' value in the failure flag denotes failure of execution, whereas a 'false' value denotes a successful execution. We use the following operations on monads:

return: $'a \Rightarrow ('s,'a)\ M$
fail: $'a \Rightarrow ('s,'a)\ M$
bind: $('s,'a)\ M \Rightarrow ('a \Rightarrow ('s,'b))\ M \Rightarrow ('s,'b)\ M$
gets: $('s \Rightarrow 'a) \Rightarrow ('s,'a)\ M$
modify: $('s \Rightarrow 's) \Rightarrow ('s, unit)\ M$

The operation *return x* does not fail, does not change the state, and returns x. The operation $fail$ sets the failure flag to true. We often use semicolon in Isabelle code for *bind*, which composes computations. The $gets$ operation applies a function to the current state and returns the result without changing the state. The $modify$ operation changes the current state using the function passed in. The code segment for monad operations is in a $do\ \cdots\ od$ block.

3 Low-level ISA Model

This section discusses the outline of our SPARCv8 ISA model. We first introduce our definition of a state, and discuss how various special features of SPARCv8 described in the previous section can be accommodated in the components of the state. We then give an example to show how an instruction is modelled. The official descriptions of SPARCv8 are sometimes semi-formal. Many details, such as memory access and cache flush, are not described at all. Thus we can only formalise those operations based on our understanding. We discuss how our formal model is validated against an actual implementation of SPARCv8, i.e., the LEON3 processor. This section closes with formal proofs of correctness and security theorems for the low-level model.

3.1 Formalisation

The core of a monadic specification is the notion of a state. Monad operations transform a state into another. The state in our SPARCv8 model is defined as:

record $('a)\ sparc\_state =$	$sys\_reg :: sys\_context$	
$cpu\_reg :: cpu\_context$	$user\_reg :: "('a)\ user\_context"$	$undef :: bool$
$mem :: mem\_context$	$mmu :: MMU\_state$	$cache :: cpu\_cache$
$dwrite :: delayed\_write\_pool$	$state\_var :: sparc\_state\_var$	$traps :: "Trap\ set"$

In general, we deal with implementation-dependent aspects of the ISA by parameterising them as variables in the model. For example, the parameter $'a$ indicates the number of windows for general registers. The *cpu_reg* are the control registers; *user_reg* are general registers; *sys_reg* are implementation-dependent system registers; followed by memory, MMU, and cache. Delayed write pool is a list of delayed write control register instructions. The state also includes necessary signals and state variables in *state_var*, which contains the annul bit, indicators of *execute_mode*, *reset_mode*, *error_mode* of the processor, among others. The state also records a set of traps (exceptions and interrupts) that may occur during execution, although in SPARCv8, there should not be more than one trap at any given time. The last member of the state is a failure flag.

The type *user_context* models windowed registers and is defined as follows:

type_synonym $window\_context = "user\_reg\_type \Rightarrow reg\_type"$
type_synonym $('a)\ window\_size = "'a\ word"$
type_synonym $('a)\ user\_context = "('a)\ window\_size \Rightarrow window\_context"$

where *user_reg_type* is a 5-bit word, *reg_type* is a 32-bit word. Our model guarantees that the global register $r[0]$ is always 0; the content of *in* registers of window n is synchronised with the content of *out* registers of window $n+1$; and the content of *out* registers of window n is synchronised with the content of *in* registers of window $n-1$. In particular, let $NWINDOWS$ be the maximum number of windows, the *in* registers of window $NWINDOWS-1$ are the same as *out* registers of window 0; *out* registers of window 0 are the same as *in* registers of window $NWINDOWS-1$.

The SPARCv8 manual does not specify how exactly memory access functions operate, it only provides interfaces for memory read and write, both of which require a memory address and an ASI as input. Accordingly, we define memory access as

type_synonym $mem\_context = "asi\_type \Rightarrow phys\_address \Rightarrow mem\_val\_type\ option"$

where *phys_address* is a 36-bit word physical address and *mem_val_type* is an 8-bit word, the length of ASI is fixed in SPARCv8 as an 8-bit word. Our model is an extension of the traditional memory access method which is usually defined as a partial function from addresses to values.

The *MMU_state* contains all the MMU registers which are used when the MMU translates a 36-bit physical address to a 32-bit virtual address by looking up three levels of Page Table Descriptors. The MMU also decides whether a page is accessible in a state or not by checking the Page Table Entry flags against the ASI. If the MMU

is turned off, the virtual address is simply translated by appending four 0s in the most significant positions (bits $32 \sim 35$ inclusive). Our MMU model conforms with the SPARCv8 reference MMU model (Appendix H, [51]).

We do not give a detailed discussion of the cache model here because it does not play an important role at the ISA level. We model it only to give information about whether the caches are empty or not, which is useful in higher level verification such as reasoning about memory context switch.

To model the delayed-write instructions, we define the following list type:

type_synonym $delayed\_write\_pool =$ "$(int \times reg\_type \times CPU\_register)\ list$"

where *int* is the delay, i.e., the number of instructions to wait. This number is reduced by 1 in every instruction execution. When the number becomes 0, the 32-bit word reg_type is written into the control register $CPU\_register$. For a write control register instruction, we add a delayed-write in the $delayed\_write\_pool$ list where the delay is implementation-dependent. If the delay is 0, the value is written to the control register immediately without modifying the pool.

We then define a $sparc\_state\_monad$ as a pair of a $sparc\_state$ and the result $'e$ of the monad:

type_synonym $('a,'e)\ sparc\_state\_monad =$ "$(('a)\ sparc\_state,'e)\ det\_monad$"

Our definition of instructions has the interface

"$instruction \Rightarrow ('a, unit)\ sparc\_state\_monad$"

where "*instruction*" is a data type consisting of the name of the instruction and all its parameters such as registers, immediates etc.

Example specification. We show an example of one of the simplest instruction formalisations here. The SETHI instruction is defined in SPARCv8 manual as below [51]:

if $(rd \neq 0)$ **then** $(r[rd]\langle 31:10\rangle \leftarrow imm22; r[rd]\langle 9:0\rangle \leftarrow 0)$

Our corresponding formalisation is given below.

```
sethi_instr instr ≡
let op_list = snd instr;
    imm22 = get_operand_w22 (op_list!0);
    rd = get_operand_w5 (op_list!1) in
if rd ≠ 0 then do
    curr_win ← get_curr_win();
    write_reg (((ucast(imm22)) :: word32) << 10) curr_win rd;
    return () od
else return ()
```

We first get the parameter *imm*22 for this instruction from op_list, which is obtained from the decoding of the instruction. To write a value into a general register, we need to get the current window, as is done by the function get_curr_win. In the SPARCv8 manual, *imm*22 is written to the bits 31 to 10 (inclusive) of *rd*, and the bits 9 to 0 are 0s. In our formalisation, we first convert the 22-bit word *imm*22 to a 32-bit word, then we shift the lower 22 bits to the left for 10 bits, leaving the lower 10 bits as 0s.

Finally, this value is written to the register by the function *write_reg*, which is defined as:

write_reg w win ur ≡ **do**
modify (λ*s*.(*user_reg_mod w win ur s*));
return () **od**

Note that the state is only changed by the modify operation. We omit details of other definitions such as *user_reg_mod*, which are available in the full formalisation in [47].

3.2 Model Execution

When executing our model, we first need to instantiate it to a particular SPARCv8 compliant processor. The LEON3 processor core [22] is a synthesisable VHDL model of a 32-bit processor compliant with the SPARCv8 ISA [51]. Its full source code is available under the GNU GPL license. We use LEON3 as a running example for our SPARCv8 model. We discuss both the execution of a single instruction and sequential composition of multiple instructions.

Exporting formal models to executable code. Before we discuss the operational semantics of instruction execution, we discuss briefly how we export our formal model into the executable code so that one can simulate instruction execution more efficiently. There has been work on exporting a formal model into executable code, e.g., [7]. However, there are various restrictions in Isabelle's code export feature; much care is required to ensure that the code can be exported. For example, Isabelle2015 cannot export a function that returns a set of functions. Consider the following example:

definition *f* :: "*int* ⇒ (*int* ⇒ *int*) *set*" **where** "*f i* ≡ {λ*x*.*x*}"

This is a legitimate definition, but the Isabelle command *value* "*f* 1", which exports the code to ML and executes it, gives an error. The original NICTA library for monad defines non-deterministic monads as below.

type_synonym ('*s*,'*a*) *nondet_monad* = "'*s* ⇒ ('*a* × '*s*) *set* × *bool*"

When we use non-deterministic monad, instruction definitions return "('*a*, *unit*) *sparc_state_monad*", which is equal to "('*a*,'*d*) *sparc_state* ⇒ (*unit* × ('*a*,'*d*) *sparc_state*) *set* × *bool*", which contains a set. The error occurs because *sparc_state* is a tuple containing functions with infinite domains. Since instruction semantics are deterministic and we do not model concurrent behaviours at the ISA level, we decide to modify the NICTA monad library to handle deterministic monads, which avoid the errors.

Single Step Execution An execution cycle in our model includes the following operations (page 158 of [51]): (1) If there is a trap, execute the trap and skip the following. (2) Execute delayed-writes. (3) Fetch and decode instruction. (4) If the annul signal is false, dispatch and execute the instruction. Then, if the instruction is not a control transfer instruction, increment program counter (PC) and next program counter (nPC) by 4. (5) If the annul signal is true, make it false, and skip this instruction.

Recall that the failure flag True in our monad means failure and False means no failure. We define a next state function as below:

"$NEXT\ s \equiv$ **case** $execute\_instruction()\ s$ **of** $(\_, True) \Rightarrow None$
$|(s', False) \Rightarrow Some\ (snd\ s')$"

We need to provide some implementation-dependent details that are not specified in the SPARCv8 model, such as the maximum number of register windows. For the LEON3 processor, we set $NWINDOWS = 8$ and $DELAYNUM = 0$, and instantiate the parameter $('a)$ in the definition of the state to a 5-bit word:

type_synonym $leon3\_state =$ "$(word\_length\ 5)\ sparc\_state$"

Finally, we need to initialise the environment, which includes PC, nPC etc., certain general registers and memory addresses that will be used in the instruction. These details will not be elaborated here, but are available from [47].

Sequential Execution. We define sequential execution as follows:

function (sequential) $SEQ ::$ "$nat \Rightarrow ('a)\ sparc\_state \Rightarrow ('a)\ sparc\_state\ option$"
where "$SEQ\ 0\ s = Some\ s$"
$|\ SEQ\ n\ s = ($**case** $SEQ\ (n-1)\ s$ **of** $None \Rightarrow None \mid Some\ t \Rightarrow NEXT\ t)$"

Preparing the environment for sequential execution requires initialising control registers and all the general registers and memory addresses involved in the sequence of instructions. We note that details such as updating PC and nPC make sequential execution easier to model and to simulate. A formal ISA model without these details may deviate from the official documentation when modeling sequential execution. Sequential execution can prove useful when analysing and validating programs.

To run large scale code such as the XtratuM hypervisor, we need to initialise the memory in our model to be consistent with real LEON3 hardware. XtratuM may assume certain values at specific memory addresses for peripheral devices etc. Performance-wise, we are able to execute an instruction in 0.005s on an Intel Xeon E5-1620 v2 CPU using a single core. Optimisation and execution of large code are left as future work.

3.3 Validation

To gain confidence that our formal model is correct, we validate our formal model against an actual implementation of SPARCv8 ISA, as described next. In the sequel, we use the OCaml version of our model extracted by the previous section. Isabelle can also generate other functional language code, but performance differences for other languages is beyond the scope of this paper.

3.3.1 Random Single Instruction Testing

Validating the formal model against real hardware by running single step instruction executions is a standard and systematic solution in the literature, cf. [20,46], to val-

idate that the formal model captures the behavour of the actual hardware it intends to model. We use a Xilinx Virtex-7 FPGA VC707 Evaluation Kit to run the official LEON3 simulator. We use the LEON3/GRLIB source code to generate bitstream code for LEON3 single core, duo core, and quad code processors. We use GRMON 2 to test the execution of instructions on those LEON3 processors. [1]

We have developed a tool to generate random instructions with random input and pre-states for our model. The random generation of instructions follows the format and op code of the SPARC instruction structure specifications. More specifically, SPARC instructions have 3 formats, cf. Fig. 2. These three formats are generated with equal probabilities. Within each format, there are sub-op codes (op2 for format 2 and op3 for format 3) that decide the exact instruction. These sub-op codes are also generated randomly in a uniform distribution. Although the chance to generate each instruction is different, with a large number of tested instances, we ensure that each of the 116 formalised instruction is tested with many different register values.

We have also written a tool to prepare the same pre-state for the LEON3 simulator, run the tests on our model and on the LEON3 simulator, and compare the results. We describe the details below.

The randomly generated instruction is checked to make sure it is a valid encoding. We then analyse the instruction instance and determine which memory addresses are involved. Our generator ensures that the majority of memory addresses are well-aligned. To initialise the pre-state, we generate random 32-bit values for the general registers in the current window and random 8-bit values for the involved memory addresses. Furthermore, we generate random flags such as the icc bits of PSR. The value of PC is 0x40000000, the values of other control registers are 0s. Since one of the intended applications of our formalisation is to reason about security properties, we also generate various tests to test integer overflow and underflow which may lead to security vulnerabilities in applications. Such tests are important to make sure that our model does not abstract away integer operations to their ideal mathematical counterparts and would thus miss potential vulnerabilities caused by integer overflow/underflow.

We then generate the GRMON 2 commands for the LEON3 simulator. The GRMON 2 commands initialise the pre-state of the LEON3 simulator to be the same as the pre-state of our model. This includes the instruction to be executed.

Our validation tool executes both our model and the LEON3 simulator, and compares the post-state. Given an instruction instance, we only examine the registers and memory addresses involved in it. The other elements in the state are not important for the validation against LEON3. For example, $delayed\_write\_pool$ is always empty. Trap set and $error\_mode$ etc. will cause exceptions and the result can be observed by the validator. The side effect of control transfer instructions (modifying the annul flag) can be checked by examining PC and nPC. The side effects of arithmetic instructions can be checked by examining PSR. Note that some of these cannot be examined in the official GRMON tool. The tested instructions should not have other side effects which may cause bugs in our model.

[1] We thank Charles Zhang for his help with our experiment setup.

Program	Number of Instructions	Time (in sec)
Addition	12	0.033
Multiplication	12	0.033
Swap two variables	14	0.041
Add the digits in a number	107	0.361
Reverse the digits in a number	116	0.339
Find the maximum number in an array	122	0.394
Greatest common divisor & least common multiple	122	0.238
Fibonacci series	141	0.468
Bubble sort	432	1.361

Table 1: Programs tested in sequential execution.

Our random testing has a large coverage. We test instructions in single core, duo core, and quad core LEON3 processors; and we test in both supervisor mode and user mode. Similarly to the validation of the ARMv7 model [20], we cannot fully test implementation-dependent system features. Our validation has tested more than 139k instruction instances. We believe our validation has been thorough and efficient, this increases our confidence of the accuracy of our model.

3.3.2 Program Execution Testing

We choose C programs that range from toy examples to non-trivial functions, covering a wide range of operations that involve most of the instructions in the IU. The programs are cross-compiled to obtain SPARC executables, from which we extract the machine code for execution. As there may be loops in the programs and it is hard to anticipate how many steps to be executed, we run the machine code on our model until we have an *instruction_access_exception* trap, which indicates that the program is finished and the next instruction is not initialised.

The tested programs are given in Table 1. The second column of Table 1 shows the number of instructions executed, the third column gives the run time in our Isabelle model. The number of instructions executed may vary depending on the input. We run these programs with arrays of length 5 for illustration. When the execution of these programs is terminated, we examine the memory addresses for the variables and arrays. Our Isabelle model gives the same result as the LEON3 simulation board on all these programs for all the tested input.

3.3.3 Limitations and Implementation-dependent Specifications for LEON3

We summarize some lessons learned from our experiment on the LEON3 board here.

According to the GRMON 2 tool, LEON3 does not implement delay write for control register instructions. Instructions such as WRPSR, WRWIM, WRY, WRTBR write the value into the register immediately. LEON3 implements 8 windows for general registers, while our SPARCv8 model supports up to 32 windows.

We approximate the LEON3 memory access behaviours by testing memory access with various ASI values: 8 (user instruction), 9 (supervisor instruction), 10 (user

data), and 11 (supervisor data) on the simulation board. We observe the following facts: (1) Writing value v to ASI 11 of address x, then reading from x in ASI 10 gives the same value v. (2) Writing v to ASI 11 of address x, then reading from x in ASI 8 gives a different value from v. (3) In both user mode and supervisor mode, reading memory with ASI 8,9,10 or 11 all work. (4) In user mode, writing to memory with ASI 11 raises a trap. (5) In user mode, writing to memory with ASI 10 will override the data at the same address in ASI 11. All the above tests assume that the MMU is turned off; otherwise, the accessibility depends on the MMU setup.

We noticed an unexpected behaviour: even in supervisor mode, writing to memory with ASI 8 or 9 does not seem to have any effect. The execution does not raise a trap, neither does it change the value at the involved addresses. This is possibly because the hardware defines the instruction memory space to be a segment of addresses we did not test. For this reason, we have only tested load/store instructions with ASI 10 and 11 in the random testing. We have enriched our SPARCv8 model with the above behaviours specific to the LEON3 processor for testing purposes. Hence our model gives the same result as the LEON3 simulation board when accessing memory in the above cases.

Due to hardware limitations, each SPARCv8 processor only accepts specific values for PSR, while our model is more general and it does not specify such details. Thus writing an arbitrary value into PSR may lead to different results in our model and in the LEON3 processor. This is not considered an error during testing. Another hardware limitation is that each board only supports a limited amount of memory, thus accessing random memory addresses may have different outcomes in our model and in the LEON3 simulator. As a result, we mainly test memory addresses ranging from 0x40000000 to 0x50000000.

The branching instructions sometimes give different results of PC and nPC when the instruction sets the "annul" bit to 1. Closer inspection reveals that the "step" command in GRMON2 may have skipped the annulled instruction and paused before the next instruction to be executed. In contrast, when we debug our simulator (an OCaml program automatically generated by Isabelle/HOL from the formal model) step by step, the OCaml debugger pauses before the annulled instruction. In this case, manual checks against the SPARCv8 manual confirm that our model is correct.

3.4 Formal Verification of Security Properties

In this subsection we prove an important security property, namely non-interference for the LEON3 processor. To give a cleaner presentation, we may simplify the Isabelle/HOL notations in the following definitions and proofs.

3.4.1 Single Step Theorem

We first show that when a state satisfies a condition called *good_context*, a single step execution from the state does not result in a failure. The execution of an instruction may generate traps, but not all traps are considered failure. A normal trap, i.e., exception or interruption, causes the CPU to run the trap handling functions, and is

not considered a failure. A failure happens only in a special situation where a trap is raised and the CPU goes to *error_mode* and awaits to be reset. The rather involved condition *good_context* is crafted to avoid failure in execution. Interested readers are referred to the source code [47] for details. We then show a single step theorem:

Theorem 1 (Single Step)
"$good\_context\ s \Longrightarrow NEXT\ s = Some\ (execute\_instruction()\ s)$"

The proof covers each instruction and shows that the monad never returns a failure if *good_context* holds; the latter is thus a good standard for verifying if a pre-state is "correct" or not.

3.4.2 Privilege Safety Theorem

Next we show that a successful one step execution in user mode does not lift the privilege to supervisor mode. *Valid* is a function obtaining the failure flag of a state.

Theorem 2 (Privilege Safety)
assumes "$get\_delayed\_pool\ s = [] \wedge\ get\_trap\_set\ s = \{\} \wedge$
$Valid\ (execute\_instruction()\ s) \wedge\ (cpu\_reg\_val\ PSR\_S\ s) = 0$"
shows "$(cpu\_reg\_val\ PSR\_S\ (execute\_instruction()\ s)) = 0$"

We assume that the delayed-write pool is empty since the LEON3 processor has no delayed write. We also assume that there are no traps to be executed. If there is a trap, the instruction will not be executed, the processor will go to supervisor mode and execute the trap instead. The third conjunct in the assumption says *execute_instruction* does not return a failure, the last conjunct says the S bit in the PSR register of the pre-state s is 0 (i.e., s is in user mode). We show that the S bit in the post-state $s' = execute\_instruction()\ s$ is also 0. This proof is a case analysis for each instruction and it checks that the execution mode is not modified.

3.4.3 Non-interference Theorem

Non-interference is an essential requirement for security. It allows user applications or virtual machines to co-exist without violating confidentiality, and it can save costly hardware which is otherwise needed to provide physical separation of data [29]. When MMU is enabled, non-interference also provides an isolation between users in different processes. That is, the high privilege resource in our setting may refer to the resource of other user processes that the current user does not have access to. This is particularly important in our project since we are interested in verifying properties for a multi-core hypervisor. Traditionally, non-interference for a deterministic program states that when a low privilege user is working on the machine, it will execute in the same manner regardless of the change of high privilege data [50]. At the ISA level, this is similar to the non-infiltration property à la Khakpour et al. [29]. Here we first show that non-interference is preserved in single step executions.

Lemma 1 (Non-interference Step)
assumes
"$(cpu\_reg\_val\ PSR\ s1) = 0 \wedge good\_context\ s1 \wedge good\_context\ s2 \wedge low\_equal\ s1\ s2$
$\wedge$
$get\_delayed\_pool\ s1 = [] \wedge get\_trap\_set\ s1 = \{\} \wedge (cpu\_reg\_val\ PSR\ s2) = 0 \wedge$
$get\_delayed\_pool\ s2 = [] \wedge get\_trap\_set\ s2 = \{\}$"
shows
"$\exists t1\ t2.Some\ t1 = NEXT\ s1 \wedge Some\ t2 = NEXT\ s2 \wedge cpu\_reg\_val\ PSR\ t1 = 0 \wedge$
$cpu\_reg\_val\ PSR\ t2 = 0 \wedge low\_equal\ t1\ t2$"

We assume that the two pre-states $s1$ and $s2$ are both in user mode, they satisfy *good_context*, they have no delayed writes and traps. We further assume that $s1$ and $s2$ are equivalent on low privilege resources. We show that the next states $t1$, $t2$ must exist, they are both in user mode, and they are still equivalent on low privilege resources. The predicate *low_equal* is defined as:

Definition 1 (*low_equal*) $low\_equal\ s1\ s2 \equiv$
$cpu\_reg\ s1 = cpu\_reg\ s2 \wedge user\_reg\ s1 = user\_reg\ s2 \wedge sys\_reg\ s1 = sys\_regs2 \wedge$
$(\forall va.(virt\_to\_phys\ va\ (mmu\ s1)\ (mem\ s1) = None \wedge$
$\quad virt\_to\_phys\ va\ (mmu\ s2)\ (mem\ s2) = None) \vee$
$\quad (\exists pa1\ pte1\ pa2\ pte2.$
$\quad\quad virt\_to\_phys\ va\ (mmu\ s1)\ (mem\ s1) = Some\ (pa1, pte1) \wedge$
$\quad\quad virt\_to\_phys\ va\ (mmu\ s2)\ (mem\ s2) = Some\ (pa2, pte2) \wedge$
$\quad\quad ((\neg(user\_writable\ pte1) \wedge \neg(user\_writable\ pte2)) \vee$
$\quad\quad\ (user\_writable\ pte1 \wedge user\_writable\ pte2)) \wedge$
$\quad\quad ((\neg(user\_readable\ pte1) \wedge \neg(user\_readable\ pte2)) \vee$
$\quad\quad\ (user\_readable\ pte1 \wedge user\_readable\ pte2 \wedge$
$\quad\quad\quad pa1 = pa2 \wedge (mem\_equal\ s1\ s2\ pa1)))\ ) \wedge$
$dwrite\ s1 = dwrite\ s2 \wedge mmu\ s1 = mmu\ s2 \wedge$
$state\_var\ s1 = state\_var\ s2 \wedge traps\ s1 = traps\ s2 \wedge undef\ s1 = undef\ s2$

Similar to Khakpour et al.'s definition, our low-equivalence assumes that the two user mode states agree on the resources that may influence the user mode execution, but we assume no knowledge about other resources. Here, *user_writable* (resp. *user_readable*) means that the flag $pte1$ related to the physical address $pa1$ is writable (resp. readable) in state $s1$. We assume that either the virtual address doesn't have a valid mapping in both states, or it has a valid mapping in both states. In the latter case, we assume that either the mapped physical addresses are both readable (similarly, writable) or both not readable (similarly, not writable). If the physical addresses are readable, then they must be identical and their content must be identical. *mem_equal* states that the block of addresses where *pa* belongs to have the same content in $s1$ and $s2$. A memory block is a group of four continuous addresses in which the first address ends with two 0s.

From the Single Step Theorem, we obtain that the one step execution from $s1$ and $s2$ will not result in failure, that is, $t1$ and $t2$ must exist. From the Safety Privilege Theorem, we know that $t1$ and $t2$ must be in user mode. The remainder of the proof

for the Non-interference Step Theorem is a case analysis for each instruction and we examine that after the execution the predicate *low_equal* holds for $t1$ and $t2$.

Finally, we show that for any sequence of user mode execution, if the initial states $s1$ and $s2$ are equivalent on low privilege resources, then the final states $t1$ and $t2$ are also equivalent on low privilege resources.

Theorem 3 (Non-interference)
assumes
"$cpu\_reg\_val\ PSR\ s1 = 0 \wedge good\_context\ s1 \wedge good\_context\ s2 \wedge low\_equal\ s1\ s2 \wedge$
$get\_delayed\_pool\ s1 = [] \wedge get\_trap\_set\ s1 = \{\} \wedge cpu\_reg\_val\ PSR\ s2 = 0 \wedge$
$get\_delayed\_pool\ s2 = [] \wedge get\_trap\_set\ s2 = \{\} \wedge user\_seq\_exe\ n\ s1 \wedge user\_seq\_exe\ n\ s2$"
shows
"$\exists t1\ t2.Some\ t1 = SEQ\ n\ s1 \wedge Some\ t2 = SEQ\ n\ s2 \wedge cpu\_reg\_val\ PSR\ t1 = 0 \wedge$
$cpu\_reg\_val\ PSR\ t2 = 0 \wedge low\_equal\ t1\ t2$"

Here, *user_seq_exe* simply assumes that every intermediate state has no traps and no delayed write instructions; these are necessary to ensure that the sequence of execution is in user mode. This proof is a simple induction on n using the Non-interference Step Theorem. The proof script of the theorems in this section measures over 7500 lines due to the large number of cases to be considered. The main difficulty is in checking that the store instructions preserve *low_equal*. This section demonstrates that we can prove interesting and non-trivial properties for SPARCv8 and LEON3 using our formalisation.

4 High-level ISA model

The previous model is suitable for reasoning about operations at instruction level, but it is too complex and detailed to reason about memory operations. Hence we simplify the low-level model by abstracting away low-level operations that are not used in the reasoning of memory operations. The operational semantics of the high-level model focuses on "memory operation blocks", which are groups of instructions, and each group contains at most one memory operation. Given the same input, the execution of an instruction in the high-level model gives the same result as the execution of the low-level model. Thus, the previous validation still holds for the high-level model.

4.1 Mapping from Instructions to Memory Operation Blocks

To bridge the gap between the instruction semantics level and the memory operation level, we define the concept of *program block* as a list of instructions where the last instruction is a memory instruction (load, store, etc.). Intuitively, each program block corresponds to a memory operation. However, sometimes there may be remaining non-memory instructions after the last memory operation, thus, we permit an exception that a program block at the end of a program may not contain any memory instruction. We illustrate program blocks with the example in Figure 4.

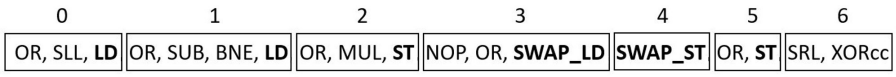

Fig. 4: Illustration of memory operation blocks.

Given a list of instructions for the processor to execute, we identify the memory access instructions (in bold font, such as LD, ST) and divide the list into several program blocks. In the example in Figure 4, there are instructions after the last memory instruction, they form a block as well (block 6), although strictly speaking block 6 is not related to a memory operation. In the SPARC TSO axiomatic model [49], an atomic load-store instruction is viewed as two memory operations $[L;S]$ where the load part L and the store part S have to be executed atomically. In correspondence, we split an atomic load-store instruction, such as SWAP, into two parts and put them in two consecutive program blocks (block 3 and 4 in Figure 4). We assume that each program block can be uniquely identified. This gives rise to a mapping $M_{block} =$ "$id \Rightarrow block$" from an identifier (natural number) to a program block. The latter is a tuple $\langle i, p, id \rangle$, where i is a list of instructions, p (natural number) is the processor in charge of executing the code, and id is a program block identifier (optional). In particular, M_{block} would map the id of the store block of an atomic load-store instruction to a triple $\langle i, p, id' \rangle$ where id' is the identifier of the load block of the same instruction.

We distinguish the types of program blocks by the memory operation involved in it. Program blocks without memory operations are called *non-mem block*, whilst program blocks including memory operations are called *memory operation blocks*. A *memory operation block* is a *load block* when it has an LD, it is a *store block* when it has an ST. An *atomic load block* has either SWAP_LD or CASA_LD, whereas an *atomic store block* has either SWAP_ST or CASA_ST.

In contrast to the low-level model, here we lift the processor execution to be oriented on program blocks, based on the *program order*. A program order is the order in which a processor executes instructions [49]. Since we can identify program blocks using their identifiers we define the program order PO for a processor p as a mapping from p to a list of identifiers: $PO =$ "$p \Rightarrow id\ list$".

Given a program order PO and a processor p, the program blocks in this program order are related by a *before* relation ";" as follows:

Definition 2 (Program Order Before) $id_1 \;;^{p}_{PO}\; id_2$ iff id_1 is before id_2 in the list of program block identifiers given by $(PO\ p)$. That is, id_1 is before id_2 in the program order if id_1 is issued by the processor p before id_2.

We shall omit the p and/or the PO in the notation of program order before and write $id_1 \;;\; id_2$ when the context is obvious. Only program blocks issued by the same processor can be related by program order. Thus $id_1 \;;\; id_2$ implicitly identifies a processor.

We divide program execution into two levels: the processors execute instructions and issue memory operations in a given program order; the memory executes memory operations in its own memory order, which will be described in Section 5.

4.2 State for Multi-core Processors

The state of a multi-core processor is a tuple $\langle ctl, reg, mem, L_{var}, G_{var}, op, undef, next \rangle$, with the following definitions:

ctl are the control registers (per processor), these include Processor State Register (PSR), which records the current set of registers, whether the processor is in user mode or supervisor mode, etc.; Program Counter (PC); Next Program Counter (nPC), among others. ctl is formally defined as a function $ctl =$ "$p \Rightarrow C_{reg} \Rightarrow val$", where p is the processor, C_{reg} is the control register, *val* is the value held by the register (32-bit word).

reg are the general registers (per processor). Formally, $reg =$ "$p \Rightarrow r \Rightarrow val$", where p is the processor, r is the address of the register (32-bit word), and *val* is the value of the register. SPARC instructions often use three general registers: two source registers, refered to as rs_1 and rs_2, and a destination register, refered to as *rd*. For instance, the addition instruction takes two values from rs_1 and rs_2, and store the sum in *rd*. We shall refer to the value *reg p rx* of a register *rx* in processor p as $r[rx]$ when the context of the processor and the state is clear. SPARC fixes the value at register address 0 to be 0. So when $rd = 0$, we have $r[rd] = 0$.

A main memory *mem* is shared by all processors. Similar to the machine code semantics for x86 [48], we focus on memory access of word (32-bits) only, and we assume that each memory address points to a word, and data are always well-aligned. Memory is a (partial) mapping $mem =$ "$addr \rightharpoonup val$". The high-level model is focused on user data memory, thus the ASI is omitted.

Each processor has a local Boolean variable $L_{var} =$ "$p \Rightarrow bool$". This Boolean variable is used to record whether the next instruction should be skipped or not after executing branching instructions. We refer to this variable as the *annul flag*.

All processors share a global variable G_{var}, which is a pair $\langle flag_{atom}, val_{rd} \rangle$, where $flag_{atom}$, defined as $id\ option$, is the id of the atomic load block when the processor is executing the corresponding atomic load-store instruction, or is undefined otherwise. val_{rd} stores the value of the general register for destination *rd* which is used in atomic load-store instructions.

op records a memory operation. Formally, $op =$ "$id \Rightarrow \langle op_{addr}, op_{val} \rangle$", where *id* is the identifier of the program block for the corresponding memory operation, op_{addr} is the address of the operation, and op_{val} is the value of the operation. For instance, a store operation writes value op_{val} at address op_{addr}, whereas a load operation loads value op_{val} from address op_{addr}. For a given *id*, op_{addr} and op_{val} are initially undefined. These values are computed during execution of memory blocks.

Finally, $undef$ indicates whether the state is undefined or not, and *next* gives the index (in the list typically given by $(PO\ p)$) of the next memory operation to be issued by processor. Formally, $next =$ "$p \Rightarrow nat$", where p is a processor and *nat* is the index.

To provide consistency w.r.t. the memory model, we split the semantics of atomic load-store instructions into the load part and the store part. The processor executes them separately, but the memory model guarantees that their executions are "atomic".

We give an example of the formalisation of the CASA instruction below. The SPARC manual [52] specifies the semantics of CASA as follows, where we adapt the setting from 64-bit registers in SPARCv9 to 32-bit registers in the SPARCv8 model:

The CASA instruction compares the register $r[rs2]$ with a memory word pointed to by the address in $r[rs1]$. If the values are equal, the value in register $r[rd]$ is swapped with the contents of the memory word pointed to by the address in $r[rs1]$. If the values are not equal, the memory location remains unchanged, but the memory word pointed to by $r[rs1]$ replace the value in $r[rd]$. We formalise the core of the load part as below, presented in pseudo-code:

Definition 3 (CASA Load) $CASA_{load}\ \ addr\ val \equiv$
if $rd \neq 0$ **then** $val_{rd} \leftarrow r[rd]$; $r[rd] \leftarrow val$; $op_{addr} \leftarrow addr$; $op_{val} \leftarrow val$;
else $val_{rd} \leftarrow r[rd]$; $op_{addr} \leftarrow addr$; $op_{val} \leftarrow val$;

Given a processor p and the *id* of a CASA load block, we can obtain the value $r[rd]$ in processor p, and the $\langle op_{addr}, op_{val}\rangle$ pair of the operation. When $rd \neq 0$, we store $r[rd]$ in the temporary global variable val_{rd}, and write val into rd. We then store $addr$ and val in op_{addr} and op_{val} respectively. When $rd = 0$, we do not have to write the rd register because its value must be 0. In this definition, $addr$ is obtained from $r[rs_1]$, and val (the value at address $addr$) is obtained from Axiom Value of the TSO model which is described in Section 5.1. The store part is given below:

Definition 4 (CASA Store) $CASA_{store}\ \ addr \equiv$
if $r[rs_2] = op_{val}$ **then** $op_{addr} \leftarrow addr$; $op_{val} \leftarrow val_{rd}$;

We check if $r[rs_2]$ has the same value as op_{val}, which corresponds to val in the load part. If this is the case, we then update op_{addr} and op_{val} with $addr$ and val_{rd} respectively, where $addr$ is the same as the address in the load part. Note that instruction semantics is only for processor execution, which does *not* update the memory. Memory write occurs in the store operation defined in the operational semantics of the TSO model, which is introduced in Section 5.2.

5 SPARC TSO Memory Model

Details of the SPARC TSO model can be found in [49,51]. This section formalises the axiomatic model in Isabelle/HOL. More importantly, we give a new operational model, which is designed to reason about execution steps and results. We show that the operational model corresponds to the axiomatic model.

5.1 Axiomatic TSO Model

The complete semantics of TSO are captured by six axioms [49,51], which specify the *ordering* of memory operations. The semantics of loads and stores to I/O addresses are implementation-dependent and are not covered by the TSO model. The SPARCv8 manual only specifies that loads and stores to I/O addresses must be strongly ordered among themselves. We adapt these axioms to our abstract SPARC ISA model and formalise them in Isabelle/HOL. Similar to the x86-TSO model [38], we focus on data memory, thus our memory model does not consider instruction fetch and flush.

Besides the *program order before* relation (cf. Definition 2), the axiomatic model also relies on a *before* relation over operations but in *memory order*, which is the order that the memory executes load and store operations. Given a partial/final memory execution in a processor i represented by a sequence x of ids, the before relation over two operations id_1 and id_2 in memory order is defined below as a partial function from the pair to $bool$, where we write $id \in x$ when id is in the sequence x:

Definition 5 (Memory Order Before) $id_1 <_x id_2 \equiv$
if $(id_1 \in x) \wedge (id_2 \in x)$ **then**
if id_1 is before id_2 in x **then** $true$ **else** $false$
else if $id_1 \in x$ **then** $true$ **else if** $id_2 \in x$ **then** $false$ **else** $undefined$
That is, id_1 is executed by the memory before id_2.

We may loosely refer to a memory order by the corresponding partial/final memory execution sequence x. We may write $id_1 < id_2$ when the context is clear. Note that any memory operation id in the sequence of executed operations x has been already executed by the processor i and thus $op_{addr}\ id$ in the current state is defined.

Due to complexity and readability reasons, we present the below definitions in prose. The axiom *Order* states that in a final execution sequence x, every pair id, id' of store operations are related by $<_x$.

Definition 6 (Axiom Order) ***Order*** $id\ id'\ x\ M_{block} \equiv$
If both $(M_{block}\ id)$ and $(M_{block}\ id')$ are either a store or an atomic store block, and both id and id' are in x, and $id \neq id'$, **then** either $(id <_x id')$ or $(id' <_x id)$.

The axiom *Atomicity* ensures that for an atomic load-store instruction, the load part id_l is executed by the memory before the store part id_s, and there can be no other store operations executed between id_l and id_s.

Definition 7 (Axiom Atomicity) ***Atomicity*** $id_l\ id_s\ PO\ x\ M_{block} \equiv$
If id_l and id_s are from the same instruction instance, and $(id_l\ ;\ id_s)$, and $(M_{block}\ id_l)$ is an atomic load block, and $(M_{block}\ id_s)$ is an atomic store block, **then** $id_l <_x id_s$, and for all store or atomic store block $(M_{block}\ id)$, if $id \in x$ and $id \neq id_s$, then either $id <_x id_l$ or $id_s <_x id$.

The axiom *Termination* states that all store operations eventually terminate. We capture this by ensuring that after the execution is completed, every store operation id that appears in the program list of some processor is in the sequence x of executed operations. We formalise this axiom as follows:

Definition 8 (Axiom Termination) ***Termination*** $id\ PO\ x\ M_{block} \equiv$
If there exists a processor p such that $id \in (PO\ p)$, and $(M_{block}\ id)$ is a store or atomic store block, **then** $id \in x$.

The axiom *Value* states that the value of a load operation id issued by processor p at address $addr$ is the value written by the most recent store to that address. The most recent store at $addr$ could be: (1) the most recent store issued by processor p, or (2) the most recent store (issued by any processor) executed by the memory.

Definition 9 (Axiom Value) ***Value*** $p\ id\ addr\ PO\ x\ M_{block}\ state \equiv$
Let S_1 be the set of store or atomic store (memory operation) blocks that are before id in the program order PO and writes to $addr$, S_2 be the set of store or atomic store blocks that are before id in the memory order $<_x$ and writes to $addr$. Let id' be the last element in the memory order in $S_1 \cup S_2$. The value to be loaded in a load block, denoted by $Lval_{id}$, is the value stored by id'.

The axiom *LoadOp* requires that any operation id' issued after a load id in the program order must be executed by the memory after id. This is formalised as below:

Definition 10 (Axiom LoadOp) ***LoadOp*** $id\ id'\ PO\ x\ M_{block} \equiv$
If $(M_{block}\ id)$ is a load or atomic load block, and $id\ ;\ id'$, **then** $id\ <_x\ id'$.

The axiom *StoreStore* states that if a store operation id is before another store operation id' in the program order, then id is before id' in the memory order.

Definition 11 (Axiom StoreStore) ***StoreStore*** $id\ id'\ PO\ x\ M_{block} \equiv$
If $(M_{block}\ id)$ and $(M_{block}\ id')$ are store or atomic store blocks, and $id\ ;\ id'$, **then** $id <_x id'$.

5.2 Operational TSO Model

Compared with other operational memory models such as the x86-TSO model [48], the high-level ISA model enables us to develop a more abstract operational memory model without using concrete modules such as the store buffer, which effectively buffers the address and value of most recent store operations. This alleviates the burden of modelling complicated operations and interactions between the processor and the store buffer, and results in a simple and elegant operational memory model. Figure 5 shows the operational semantics of the TSO model, which are described below. The function exe_{id} executes all the instructions in the operation block identified by id, exe^{pre}_{id} executes the block until (but not including) the last instruction in the block, exe^{last}_{id} executes only the last instruction in the block. The formal semantics for these functions are very tedious and we omit them here. We refer the reader to the Isabelle/HOL theories [47] for the details.

We shall use the following notations: We write $type_{id}$ to denote the type of the memory operation block $(M_{block}\ id)$. We use the following abbreviations for memory operation block types: ld (load), ald (atomic load), st (store), ast (atomic store), non (non-mem). We write $x@x'$ for the concatenation of two sequences x and x'.

The operational TSO model consists of four operation rules, which are explained below. Each rule adds the conditions that must be satisfied to apply the rule as *premises*. A rule also has a sequence of *operations*, which takes a pair $\langle x, s\rangle$ of a memory operation sequence x and state s as input, and returns a new sequence x' and a new state s' as output. We write the overall operation of a rule as $x, s \rightsquigarrow x', s'$. We often use "execute until the last instruction in a block", because the last instruction is defined as the memory operation instruction (cf. Figure 4).

$$\frac{type_{id} = ld \qquad \forall id'.\ ((id'\ ;\ id)\ \wedge\ type_{id'} \in \{ld, ald\} \longrightarrow\ id' \in x)}{x, s \rightsquigarrow x@[id], (exe_{id}^{last}\ Lval_{id}\ (exe_{id}^{pre}\ s))}\ load$$

$$\frac{type_{id} = st \qquad flag_{atom} = undefined \qquad \forall id'.((id'\ ;\ id)\ \wedge\ type_{id'} \in \{ld, ald, st, ast\} \longrightarrow id' \in x)}{x, s \rightsquigarrow x@[id], (W_{mem}\ id\ (exe_{id}\ s))}\ store$$

$$\frac{type_{id} = ald \qquad flag_{atom} = undefined \qquad \forall id'.((id'\ ;\ id)\ \wedge\ type_{id'} \in \{ld, ald, st, ast\} \longrightarrow id' \in x)}{x, s \rightsquigarrow x@[id], (flag_{atom}^{set}\ id\ (exe_{id}^{last}\ Lval_{id}\ (exe_{id}^{pre}\ s)))}\ atom\_load$$

$$\frac{type_{id} = ast \qquad flag_{atom} = id' \qquad atom_{pair}\ id = id' \qquad \forall id''.((id''\ ;\ id)\ \wedge\ type_{id''} \in \{ld, ald, st, ast\} \longrightarrow id'' \in x)}{x, s \rightsquigarrow x@[id], (W_{mem}\ id\ (flag_{atom}^{set}\ undef\ (exe_{id}\ s)))}\ atom\_store$$

Fig. 5: Rules for the operational TSO model.

The load rule is defined as below:

Premises :
- $type_{id} = ld$.
- $\forall id'.\ ((id'\ ;\ id)\ \wedge\ type_{id'} \in \{ld, ald\} \longrightarrow\ id' \in x)$.

Operation :
- $x' = x@[id]$.
- From state s, execute the block id until the last instruction, obtain s_1.
- From s_1, get load value via $Lval_{id}$ (cf. Definition 9).
- From s_1, execute the last instruction in the block id, obtain s'.

The store rule is defined as below:

Premises :
- $type_{id} = st$.
- $flag_{atom} = undefined$.
- $\forall id'.((id'\ ;\ id)\ \wedge\ type_{id'} \in \{ld, ald, st, ast\} \longrightarrow id' \in x)$.

Operation :
- $x' = x@[id]$.
- From state s, execute the block id, obtain s_1.
- From s_1, write the value into memory, obtain s'.

The atom_load rule is defined as below:

Premises :
- $type_{id} = ald$.
- $flag_{atom} = undefined$.
- $\forall id'.((id'\ ;\ id)\ \wedge\ type_{id'} \in \{ld, ald, st, ast\} \longrightarrow id' \in x)$.

Operation :

- $x' = x@[id]$.
- From state s, execute the block id until the last instruction, obtain s_1.
- From s_1, get load value via $Lval_{id}$ (cf. Definition 9).
- From s_1, execute the last instruction in the block id, obtain s_2.
- From s_2, set atomic flag $flag_{atom}$ to id, obtain s'.

The atom_store rule is defined as below:

Premises :
- $type_{id} = ast$.
- $flag_{atom} = id'$.
- id' is the atomic load block in the same instruction as id (similar to SWAP_LD and SWAP_ST in Figure 4).
- $\forall id''.((id'' \ ; \ id) \ \wedge \ type_{id''} \in \{ld, ald, st, ast\} \longrightarrow id'' \in x)$.

Operation :
- $x' = x@[id]$.
- From state s, execute the block id, obtain s_1.
- From s_1, set atomic flag $flag_{atom}$ to $undefined$, obtain s_2.
- From s_2, write the value in memory, obtain s'.

In addition to the rules for memory operations, to obtain the final result of processor execution, we may need the *non_mem* rule, because there may be instructions after the last memory operation instruction (cf. Figure 4). This rule is defined as below:

Premises :
- $type_{id} = non$.
- $\forall id'.((id' \ ; \ id) \ \wedge \ type_{id'} \in \{ld, ald, st, ast\} \longrightarrow id' \in x)$.

Operation :
- $x' = x@[id]$.
- From state s, execute the block id, obtain s'.

The *non* rule is not related to the memory model because it does not involve memory operations. It plays no roles in the proofs in the remainder of this section.

5.3 Soundness and completeness of the operational model

Theorem 4 (Soundness) *Every memory operation sequence generated by the operational model satisfies the axioms in the axiomatic model.*

Theorem 5 (Completeness) *Every memory operation sequence that satisfies the axioms in the axiomatic model can be generated by the operational model.*

(Proof outline:) The previous subsection has briefly discussed that the design of operational rules respects the axioms such as LoadOp, StoreStore, and Atomicity. Axiom Value trivially holds in the operational model because the rule *load* directly uses axiom Value to obtain load result. Axiom Termination is satisfied by the construction of the execution witness sequences, because the x part of the final witness is guaranteed to contain all the store operations, which means that the execution of

Processor	*op_id*	Instruction
1	0	$OR \quad \%g0, 1, \%r4$ $OR \quad \%g0, 1, \%r5$ $ST \quad \%r5, [\%g0 + \%r4]$
	1	$OR \quad \%g0, 1, \%r5$ $OR \quad \%g0, 2, \%r4$ $ST \quad \%r5, [\%g0 + \%r4]$
2	2	$OR \quad \%g0, 2, \%r4$ $LD \quad [\%g0 + \%r4], \%r1$
	3	$OR \quad \%g0, 3, \%r4$ $ST \quad \%r1, [\%g0 + \%r4]$
3	4	$OR \quad \%g0, 3, \%r4$ $LD \quad [\%g0 + \%r4], \%r1$
	5	$OR \quad \%g0, 1, \%r4$ $LD \quad [\%g0 + \%r4], \%r2$

Table 2: "Indirection Through Processors".

these operations have been completed by the memory. Axiom Order holds because all the executed store operations are recorded in a list, which means every pair of them are ordered. The completeness proof is an induction on the length of the memory operation sequence. It essentially shows that, given a partial execution, appending any valid memory operation to the execution sequence will result in a new sequence that can be executed by the operational model. The formal proof of the correspondence of the axiomatic model and the operational model is rather complicated, interested readers can check the Isabelle/HOL formalization and proofs[2] for more details.

5.4 Case Studies

We can now formally reason about concurrent machine code. The axiomatic model can be used to reason about the order of memory operations, while the operational model is better at reasoning about properties of the execution flow. We run two case studies drawn from examples in the SPARCv9 manual [52]. We may use the term *process* and *processor* interchangeably. See Owen's work [37] for a semantic foundation for reasoning about programs in TSO-like relaxed memory models.

5.4.1 Indirection Through Processors

The "Indirection Through Processors" program is taken from Figure 46 of the SPARCv9 manual [52]. This example intends to reflect the TSO property that causal update relations are preserved. The original program involves three processors, each processor issues two memory operations. A memory operation is given in an "instruction-like" style, e.g., $st \quad \#1, [A]$ means that the value 1 is stored into address A of the memory. Unfortunately in real SPARC store instructions, the value to be stored and the value of the memory address must be taken from registers, so we need to add a few

[2] Isabelle/HOL code for proofs is at https://github.com/CompSoftVer/SPARCv8-Models

instructions to initialise the registers for this example to work. Our formalised "Indirection Through Processors" example is shown in Table 2. The global register $\%g0$ in SPARC always contains 0. The first instruction in block 0 adds 0 and 1, and puts the result in register $\%r4$. The *ST* in block 0 thus stores 1 at memory address 1. The *ST* in block 1 stores 1 at address 2. The *LD* in block 2 loads the value at address 2 to register $\%r1$. Block 3 then stores the value in $\%r1$ at address 3. Finally, processor 3 loads the values at addresses 3 and 1 to registers $\%r1$ and $\%r2$.

Reasoning about memory operation order. It is intuitive to use the axiomatic TSO model to reason about the order of memory operations. For the program in Table 2, the SPARCv9 manual gives some example sequences of memory operations allowed under TSO, and an example sequence that is not allowed under TSO: $x = [1,2,3,4,5,0]$. This is because $(0\ ;\ 1)$ must hold in the program order given by Table 2, and the above sequence implies that $\neg(0 < 1 = true)$ in the memory order, which falsifies the axiom StoreStore.

Alternatively, the completeness of the operational TSO model enables us to use the operational model to reason about the possible next step operations. The above reasoning can be confirmed by our operational model in the lemma below:

Lemma 2 $init \longrightarrow \neg([], s \rightsquigarrow [1], s')$

Lemma 2 states that given a partial execution sequence which contains only an initialisation step *init* where memory addresses are set to $undefined$ and registers are set of 0, memory operation block 1 in Table 2 cannot be the first operation to be executed.

Reasoning about execution result. Besides eliminating illegal executions, one can also use our operational model to reason about the results of legal executions. For instance, the SPARCv9 manual lists the sequence $x' = [0,1,2,3,4,5]$ as a legal execution under TSO. For simplicity, here we only show that after a partial execution $[0,1,2]$, the register $\%r1$ of processor 2 has value 1, which is stored to address 2 by processor 1 previously. This shows that a processor can observe the memory updates made by other processors. This is formalised in the following lemma:

Lemma 3 $[0,1], s_2 \rightsquigarrow [0,1,2], s_3 \longrightarrow (reg\ s_3)\ 2\ 1 = 1$

The right hand side of the implication means that in state s_3, the general register 1 of processor 2 contains value 1. The proof for execution results usually involves a "simulation" of the execution using the abstract ISA model and the operational TSO model. For this example, we start from the initial witness, and prove a series of lemmas about the execution witnesses $([0], s_1), ([0,1], s_2), ([0,1,2], s_3)$ for the intermediate execution steps. It is straightforward to complete this series of proofs and obtain the result of a final execution.

Lock$(lock, proc\_id)$
retry:
  mov $[proc\_id], \%l0$
  cas $[lock], \%g0, \%l0$
  tst $\%l0$
  be out
  nop
loop:
  ld $[lock], \%l0$
  tst $\%l0$
  bne loop
  nop
  ba,a retry
out:
  code in critical region

Unlock$(lock)$
  st $\%g0, [lock]$

(a) Spin lock using CASA.

Processor	op_id	Instruction
1	0	$OR \quad \%g0, 1, \%r16$ $OR \quad \%g0, 1, \%r1$ $CASA\_LD \quad [\%r1], \%g0, \%r16$
	1	$CASA\_ST \quad [\%r1], \%g0, \%r16$
	5	$ORcc \quad \%g0, \%r16, \%g0$ $BE \quad 28$ NOP
2	2	$OR \quad \%g0, 1, \%r16$ $OR \quad \%g0, 1, \%r1$ $CASA\_LD \quad [\%r1], \%g0, \%r16$
	3	$CASA\_ST \quad [\%r1], \%g0, \%r16$
	6	$ORcc \quad \%g0, \%r16, \%g0$ $BE \quad 28$ NOP
3	4	$OR \quad \%g0, 1, \%r4$ $ST \quad \%g0, [\%g0 + \%r4]$

(b) A fragment of formalised spin lock code.

Fig. 6: The spin lock example.

5.4.2 Spin Lock with Compare and Swap

Section J.6 of the SPARCv9 manual [52] gives an example of spin lock implemented using the CASA instruction, the code is shown in Figure 6a. Note that the code in Figure 6a is in *synthetic instruction* format. SPARCv8/v9 manual provides a straightforward mapping from this format to *SPARC instruction* format, which is what our ISA model supports. For instance, in the *retry* fragment, the first instruction *mov* corresponds to an *OR*, which adds the ID *proc_id* of the current process and 0, and stores the result to register $\%l0$, which corresponds to register $\%r16$. After executing this line, $\%l0$ ($\%r16$) contains the ID of the current process. The second line is the CASA instruction. It checks whether the memory value at address $lock$ is equal to the value at $\%g0$ (which must be 0), and swaps the value at address $lock$ and the value at register $\%l0$ when the above check is positive. Otherwise, the value at address $lock$ is stored at register $\%l0$. Therefore, when no processes hold the lock, the value at address $lock$ is 0, and after executing the second line, $\%l0$ ($\%r16$) will have 0 and address $lock$ will contain the ID of the current process. On the other hand, when the lock is held by another process, after executing CASA, the memory address $lock$ is unchanged, and $\%l0$ contains the ID of the process that holds the lock. The code $tst \ \ \%l0$ corresponds to an *ORcc*, which checks if $\%l0$ is equal to 0. If it is, then the program branches to *out*, and starts to execute in the critical region. Otherwise, the program goes to $loop$ and keeps reading the address $lock$ until it contains a 0.

We give the fragment of instructions before entering the critical region in Figure 6b, and consider a concrete situation where two processes (processors) 1 and 2 are competing to get the lock, and process 3 initialises the lock to 0. Assume that process 3 executes operation 4 first for initialisation, also assume without of loss generality that operation 0 of process 1 is executed by the memory earlier than process

2's operations, we show that process 1 will enter the critical region. The case where operation 2 of process 2 is executed earlier by the memory is symmetric. In this example, we set the address of critical region as $28 \ll 2 = 112$ relative to the address of the branch instruction *BE* (Branch if Equal), where $\ll$ is *sign extended shift to the left*.

The proof uses a mixture of the techniques in the previous subsection to obtain valid memory operation sequences and reason about the results. We omit the intermediate steps and show the final lemma below:

Lemma 4 $[4,0,1,2,3,5], s_6 \rightsquigarrow [4,0,1,2,3,5,6], s_7 \longrightarrow (ctl\ s_7)\ 1\ nPC = (ctl\ s_7)\ 1\ PC + 112 \wedge (ctl\ s_7)\ 2\ nPC = (ctl\ s_7)\ 2\ PC + 4$

The right hand side of the implication shows that the *nPC* (next program counter) of processor 1 is the entry point of the critical region, while the *nPC* of processor 2 points to *NOP*, after which will lead processor 2 to the loop in Figure 6a.

6 Conclusion

This paper describes a formalisation of the SPARC ISA and TSO memory model. The low-level ISA model has over 5000 lines of Isabelle code, not including the proofs. The high-level ISA model measures 1960 lines of code, the two memory models and the soundness and completeness proofs constitute 4753 lines of code, the case studies take up 1750 lines of code.

The formal model can be specialised to any SPARCv8 processor, and it contains many features specific to the SPARCv8 architecture. We have validated the low-level model against an official LEON3 simulator on more than 100k random instruction instances as well as real life programs. To illustrate the applicability of our model, we have shown a non-interference property for the LEON3 processor. This property guarantees that user mode execution is independent of high privilege resources which the user has no access to.

On top of the low-level ISA model, we formalise the high-level ISA model and the SPARC TSO axiomatic memory model in Isabelle/HOL. This model is useful for reasoning about the order of memory operations. We also give a new operational TSO memory model as a system that consists of four rules. We show that the operational TSO model is sound and complete with respect to the axiomatic model. Finally, we demonstrate the use of our memory models with two examples in the SPARCv9 manual.

Our current on-going work is about developing a Hoare-style logic for SPARC machine code. The current framework, which includes the abstract ISA model and the memory models, provides the foundation for the verification of concurrent machine code. However, if a program involves a complex control-flow with branches and loops, it is tedious to use the current models to reason about the program. A Hoare-style logic is much desired to make the reasoning task easier. We envision that this new work will make it easier to prove properties such as reachability, safety, and non-interference.

Acknowledgement. This work has been partially supported by the National Satellite of Excellence in Trustworthy Software Systems (Award No. NRF2018NCR-NSOE003), and award NRF Investigatorship NRFI06-2020-0022,funded by NRF Singapore under National Cyber-security R&D (NCR) programme.

References

1. L3 specification language for ISAs. `http://www.cl.cam.ac.uk/~acjf3/l3/`. [Online; accessed 09/12/2015].
2. RISC-V architecture. `https://riscv.org/`. [Online; accessed 10/08/2016].
3. Tianhe-2. `http://top500.org/system/177999`. [Online; accessed 27/01/2016].
4. J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. *Fences in Weak Memory Models*, pages 258–272. Springer Berlin Heidelberg, 2010.
5. J. Alglave, L. Maranget, and M. Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.*, 36(2), July 2014.
6. D. Aspinall and J. Ševčík. *Formalising Java's Data Race Free Guarantee*, pages 22–37. Springer Berlin Heidelberg, 2007.
7. R. Atkey. CoqJVM: An executable specification of the Java virtual machine using dependent types. In *TYPES*, LNCS, pages 18–32. Springer, 2005.
8. G. Boudol and G. Petri. Relaxed memory models: an operational approach. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 392–403, 2009.
9. S. Burckhardt and M. Musuvathi. *Effective Program Verification for Relaxed Memory Models*, pages 107–120. Springer Berlin Heidelberg, 2008.
10. B. Campbell and I. Stark. Randomised testing of a microprocessor model using SMT-solver state generation. In *FMICS 2014*, pages 185–199. Springer, 2014.
11. D. Cock, G. Klein, and T. Sewell. Secure microkernels, state monads and scalable refinement. In *Theorem Proving in Higher Order Logics*, volume 5170 of *LNCS*, pages 167–182. Springer, 2008.
12. K. Crary and M. J. Sullivan. A calculus for relaxed memory. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 623–636. ACM, 2015.
13. S. Dasgupta, D. Park, T. Kasampalis, V. S. Adve, and G. Roşu. A complete formal semantics of x86-64 user-level instruction set architecture. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 11331148, New York, NY, USA, 2019. Association for Computing Machinery.
14. S. El Kady, M. Khater, and M. Alhafnawi. MIPS, ARM and SPARC-an architecture comparison. In *Proceedings of the World Congress on Engineering*, volume 1, 2014.
15. ESA. ESA LEON processor. `http://www.esa.int/Our_Activities/Space_Engineering_Technology/LEON_the_space_chip_that_Europe_built`, 2017. [Online; accessed 19/06/2016].
16. S. Flur, K. E. Gray, C. Pulte, S. Sarkar, A. Sezgin, L. Maranget, W. Deacon, and P. Sewell. Modelling the armv8 architecture, operationally: Concurrency and ISA. *SIGPLAN Not.*, 51(1):608–621, Jan. 2016.
17. A. Fox. Formal specification and verification of ARM6. In *Theorem Proving in Higher Order Logics*, volume 2758 of *LNCS*, pages 25–40. Springer, 2003.
18. A. Fox. Directions in ISA specification. In *Interactive Theorem Proving*, volume 7406 of *LNCS*, pages 338–344. Springer Berlin Heidelberg, 2012.
19. A. Fox. Improved tool support for machine-code decompilation in HOL4. In *Interactive Theorem Proving 2015*, pages 187–202, 2015.
20. A. Fox and M. O. Myreen. A trustworthy monadic formalization of the ARMv7 instruction set architecture. In *Interactive Theorem Proving*, pages 243–258, 2010.
21. Fujitsu. K computer. `http://www.top500.org/system/177232`, 2017. [Online; accessed 19/06/2016].
22. Gaisler. LEON3 processor. `http://www.gaisler.com/index.php/products/processors/leon3`, 2017. [Online; accessed 19/06/2017].

23. S. Goel. Formal verification of application and system programs based on a validated x86 isa model, 2016. PhD Thesis, The University of Texas at Austin.
24. S. Goel, W. A. Hunt, and M. Kaufmann. Abstract stobjs and their application to ISA modeling. In *ACL2 2013*, pages 54–69, 2013.
25. K. E. Gray, G. Kerneis, D. Mulligan, C. Pulte, S. Sarkar, and P. Sewell. An integrated concurrency and core-ISA architectural envelope definition, and test oracle, for IBM POWER multiprocessors. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 635–646. ACM, 2015.
26. S. Hangal, D. Vahia, C. Manovit, and J.-Y. J. Lu. Tsotool: A program for verifying memory systems using the memory consistency model. *SIGARCH Comput. Archit. News*, 32(2):114–, 2004.
27. L. Higham, J. Kawash, and N. Verwaal. Defining and comparing memory consistency models. In *In Proc. of the 10th Int'l Conf. on Parallel and Distributed Computing Systems*, pages 349–356, 1997.
28. Z. Hou, D. Sanán, A. Tiu, Y. Liu, and K. C. Hoa. An executable formalisation of the sparcv8 instruction set architecture: A case study for the LEON3 processor. In *FM 2016: Formal Methods - 21st International Symposium, 2016, Proceedings*, pages 388–405, 2016.
29. N. Khakpour, O. Schwarz, and M. Dam. Machine assisted proof of ARMv7 instruction level isolation properties. In *Certified Programs and Proofs*, volume 8307, pages 276–291. LNCS, 2013.
30. X. Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *In Proceedings. 33rd ACM Symposium on Principles of Programming Languages*, 2006.
31. X. Leroy. The CompCert C verified compiler. `http://compcert.inria.fr/man/manual.pdf`, 2015. [Online; accessed 29/01/2016].
32. J. Lim and T. Reps. Tsl: A system for generating abstract interpreters and its application to machine-code analysis. *ACM Trans. Program. Lang. Syst.*, 35(1), Apr. 2013.
33. H. Liu and J. S. Moore. Executable JVM model for analytical reasoning: A study. In *Proceedings of the 2003 Workshop on Interpreters, Virtual Machines and Emulators*, pages 15–23. ACM, 2003.
34. P. Loewenstein and S. Chaudhry. Multiprocessor memory model verification. In *Proc. Automated Formal Methods*. FLoC workshop, 2006.
35. D. Lustig, M. Pellauer, and M. Martonosi. Pipecheck: Specifying and verifying microarchitectural enforcement of memory consistency models. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 635–646, Los Alamitos, CA, USA, dec 2014. IEEE Computer Society.
36. D. P. Mulligan, S. Owens, K. E. Gray, T. Ridge, and P. Sewell. Lem: reusable engineering of real-world semantics. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, pages 175–188, 2014.
37. S. Owens. Reasoning about the implementation of concurrency abstractions on x86-tso. In *Proceedings of the 24th European Conference on Object-oriented Programming*, ECOOP'10, pages 478–503, 2010.
38. S. Owens, S. Sarkar, and P. Sewell. *A Better x86 Memory Model: x86-TSO*, pages 391–407. Springer Berlin Heidelberg, 2009.
39. S. Park and D. L. Dill. An executable specification, analyzer and verifier for rmo (relaxed memory order). In *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '95, pages 34–41. ACM, 1995.
40. G. Petri. Operational semantics of relaxed memory models, 2010. Thesis.
41. C. Pulte, S. Flur, W. Deacon, J. French, S. Sarkar, and P. Sewell. Simplifying arm concurrency: Multicopy-atomic axiomatic and operational models for armv8. *Proc. ACM Program. Lang.*, 2(POPL), Dec. 2017.
42. G. Roşu and T. F. Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.
43. G. Roşu and A. Ştefănescu. Towards a unified theory of operational and axiomatic semantics. In A. Czumaj, K. Mehlhorn, A. Pitts, and R. Wattenhofer, editors, *Automata, Languages, and Programming*, pages 351–363, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
44. A. Roy, S. Zeisset, C. J. Fleckenstein, and J. C. Huang. *Fast and Generalized Polynomial Time Memory Consistency Verification*, pages 503–516. Springer Berlin Heidelberg, 2006.
45. A. Santoro, W. Park, and D. Luckham. SPARC-V9 architecture specification with Rapide. Technical report, Stanford, CA, USA, 1995.
46. S. Sarkar, P. Sewell, F. Z. Nardelli, S. Owens, T. Ridge, T. Braibant, M. O. Myreen, and J. Alglave. The semantics of x86-CC multiprocessor machine code. In *Proceedings of the 36th Annual ACM Symposium on Principles of Programming Languages*, pages 379–391. ACM, 2009.

47. Securify. Securify: Micro-kernel verification. http://securify.scse.ntu.edu.sg/MicroVer/. [Online; accessed 20/03/2020].
48. P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen. X86-tso: A rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7):89–97, July 2010.
49. P. S. Sindhu, J.-M. Frailong, and M. Cekleov. *Formal Specification of Memory Models*, pages 25–41. Springer US, Boston, MA, 1992.
50. G. Smith. Principles of secure information flow analysis. In *Malware Detection*, pages 291–307, 2007.
51. SPARC. The SPARC architecture manual version 8. http://gaisler.com/doc/sparcv8.pdf, 1992. [Online; accessed 27/10/2015].
52. SPARC. The SPARC architecture manual version 9. https://cr.yp.to/2005-590/sparcv9.pdf, 1994. [Online; accessed 12/06/2017].
53. XtratuM. Xtratum hypervisor. http://www.xtratum.org/, 2017. [Online; accessed 19/06/2017].
54. Y. Yang, G. Gopalakrishnan, G. Lindstrom, and K. Slind. Nemos: a framework for axiomatic and executable specifications of memory consistency models. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, April 2004.
55. Y. Zhao, D. Sanán, F. Zhang, and Y. Liu. Reasoning about information flow security of separation kernels with channel-based communication. In *TACAS 2016*, volume 9636, pages 791–810. Springer, 2016.