An Executable Formalisation of the SPARCv8 Instruction Set Architecture: A Case Study for The LEON3 Processor

Zhe Hou[1], David Sanan[1], Alwen Tiu[1], Yang Liu[1], and Koh Chuen Hoa[2]

[1] School of Computer Science and Engineering, Nanyang Technological University
[2] Singapore DSO

Abstract. The SPARCv8 instruction set architecture (ISA) has been used in various processors for workstations, embedded systems, and space missions. However, there are no publicly available formal models for the SPARCv8 ISA. In this work, we give the first formal model for the integer unit of SPARCv8 ISA in Isabelle/HOL. We capture the operational semantics of the instructions using monadic definitions. Our model is a detailed model, which covers many features specific to SPARC processors, such as delayed-write for control registers, windowed general registers, and more complex memory access. Our model is also general, as we retain an abstract layer of the model which allows it to be instantiated to support all SPARCv8 compliant processors. We extract executable code from our formalisation, giving us the first systematically verified executable semantics for the SPARCv8 ISA. We have tested our model extensively against a LEON3 simulation board, covering both single-step executions and sequential execution of programs. We prove some important properties for our formal model, including a non-interference property for the LEON3 processor.

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 [20]. There have been various publicly available formal models for ISAs in the literature, e.g., for ARM6 [14], ARMv7 [17], x86 [25]. However, to the best of our knowledge, there are no formalisations of the SPARC family architectures.

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 [13]. SPARC is also used in supercomputers. Fujitsu's K computer [2], ranked NO.1 in TOP500 2011, combined 88,128 SPARC CPUs. Tianhe-2 [8], 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. ESA chose to use SPARCv8, mainly because SPARC is one of the few fully open ISAs (other than RISC-V [5] etc.), and has significant support. ESA then started the LEON project to develop processors for space projects [1].

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 formalize each layer separately and use a refinement-based approach to show that important 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 formalization of the high-level specification of a separation micro-kernel [27], 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 formalize the XtratuM [9] micro-kernel that runs on top of the multi-core LEON3 processor; these formalization efforts are still on-going. The ISA formalisation described in this paper is a key component bridging these two formalizations. 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 formalizing the integer unit (IU) of SPARCv8, which contains all the instructions used in XtratuM.

This paper presents the first detailed Isabelle/HOL model for the IU in the SPARCv8 ISA. Although there are formal models for other ISAs in the literature (e.g., [14,25]), 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 [13]. 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 [3].

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 [7]. We also provide a memory management unit (MMU) model to support multi-core micro-kernel 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.

Our main contributions are: (1) We give a formal model for the IU of SPARCv8 ISA. (2) Our model can be exported to OCaml code for both single step execution

and sequential execution. (3) Our model has been extensively tested against a LEON3 simulation board through more than 100,000 instruction instances. (4) To demonstrate the applicability of our model, we first prove a correctness property which ensures that the execution of an instruction will not result in failure when the pre-state satisfies a well-formedness condition. We also show a security property: if the pre-state meets certain conditions, then the privilege will not be lifted during the execution. Finally, we show a non-interference property for the LEON3 processor: given two user-mode states which have the same low privilege resources, after a series of user-mode execution, the low privilege resources in the two resultant states are still equivalent. That is, the difference in high privilege data does not affect low privilege execution.

The complete source code of our formalization of SPARCv8 ISA and the simulator extracted from our formal model can be found at the Securify project website [6].

Related work. Santoro et al. [24] gave an executable specification for the SPARCv9 architecture with Rapide. However, their model is not built in a theorem proving, thus it is not suitable for formal verification purposes. Fox studied verification of the ARM6 micro-architecture at the RTL level [14]. 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 [17]. 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 [15,3]. Fox recently developed a framework for formal verification of ISAs [16]. 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 [17] and [11]. There are also formal models for the x86 architecture, such as Sarkar et al.'s work on the semantics of x86-CC machine code [25]. Another interesting work is the ACL2 ISA models [18]. 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. 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 [21,22]. A remotely related work is Liu and Moore's executable JVM model M6 [23], 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 [10] inspired us to define the model in a proof assistant which supports code export for execution.

2 Background

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

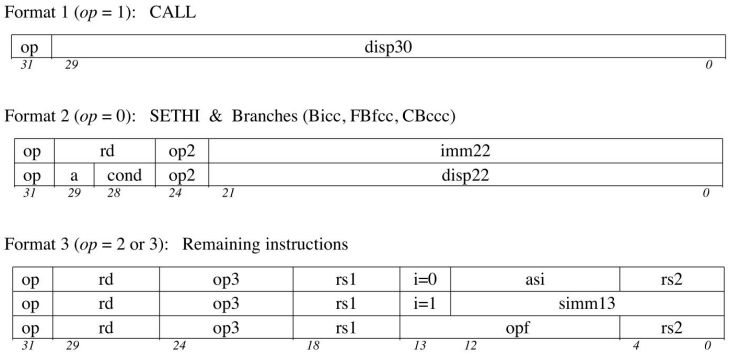

Fig. 1. The formats for SPARCv8 instructions. Source: [7].

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. 1. 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 vectored 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 SPARC are grouped in overlapping windows. This design allows for straightforward, high-performance compilers and a significant reduction in memory load/store instructions over other RISCs [7]. 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 register 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. This is visualised in Fig. 2. The windows are arranged in a circular buffer, where the last window's *out registers* overlaps with the first window's *in* registers. The current window of registers 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

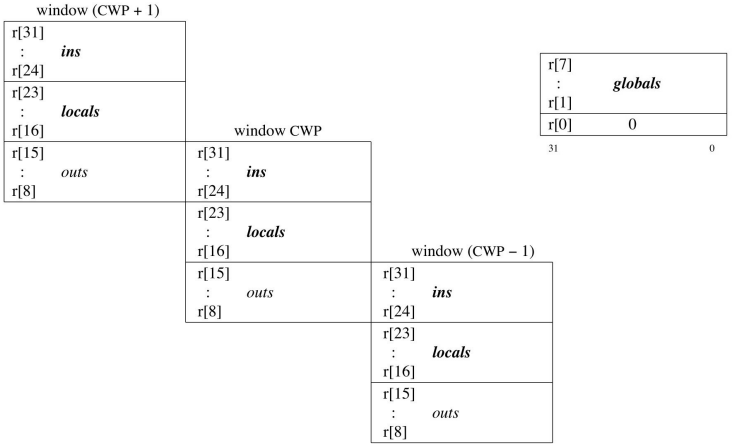

Fig. 2. Three overlapping windowed registers and the global registers. Source: [7].

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 [7].

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" [7].

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 [17], 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 [12]. Instead of using non-deterministic monads in [12], here we use deterministic monads (cf. Section 4 for reasons) defined as below, where `M` is a shorthand for `det_monad`.

```
type_synonym ('s, 'a) M = "'s ⇒ ('a × 's) × bool"
```

which returns a pair `('a × '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 ⇒ ('s, 'a) M
fail: 'a ⇒ ('s, 'a) M
bind: ('s, 'a) M ⇒ ('a ⇒ ('s, 'b) M) ⇒ ('s, 'b) M
gets: ('s ⇒ 'a) ⇒ ('s, 'a) M
modify: ('s ⇒ 's) ⇒ ('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 ··· od` block.

3 Isabelle/HOL Specification for the SPARCv8 ISA

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. In Section 5 we discuss how our formal model is validated against an actual implementation of SPARCv8, i.e., the LEON3 processor.

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 =
cpu_reg:: cpu_context          user_reg:: "('a) user_context"
sys_reg:: sys_context          mem:: mem_context
mmu:: MMU_state                cache:: cpu_cache
dwrite:: delayed_write_pool    state_var:: sparc_state_var
traps:: "Trap set"             undef:: bool
```

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 ⇒ reg_type"
type_synonym ('a) window_size = "'a word"
type_synonym ('a) user_context = "('a) window_size ⇒ 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 ⇒ phys_address ⇒ 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 two 0s in the beginning. Our MMU model conforms with the SPARCv8 reference MMU model (Appendix H, [7]).

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 × reg_type × 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

```
"('b) instruction ⇒ ('a,unit) sparc_state_monad"
```

where `"('b) 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 [7]:

```
if (rd ≠ 0) then (r[rd]<31:10> ← imm22;r[rd]<9:0> ← 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 `imm22` 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, `imm22` 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 `imm22` 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 (λs.(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 formalization in [6].

4 Model Execution

When executing our model, we first need to instantiate it to a particular SPARCv8 compliant processor. The LEON3 processor core [4] is a synthesisable VHDL model of a 32-bit processor compliant with the SPARCv8 ISA [7]. 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., [10]. 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 ≡ {λx. 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 [7]): (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 ≡ case execute_instruction() s of (_,True) ⇒ None
  | (s',False) ⇒ 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_length5) 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 [6].

Sequential Execution. We define sequential execution as follows:

```
function (sequential) SEQ:: "nat ⇒ ('a) sparc_state ⇒ ('a)
    sparc_state option" where "SEQ 0 s = Some s"
|"SEQ n s = (case SEQ (n-1) s of None ⇒ None | Some t ⇒ 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.

5 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.

5.1 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. [17,25], to gain confidence 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. [3]

We have developed a tool to generate random instructions with random input and pre-states for our model. 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

[3] 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.

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.

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 [17], we cannot fully test implementation-dependent system features. Our validation has tested more than 100k instruction instances, and still counting. We believe our validation has been thorough and efficient; this increases our confidence of the accuracy of our model.

5.2 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 various input.

5.3 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. If the MMU is turned on, then 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, whereas our model pauses before the annulled instruction. In this case, manual checks against the SPARCv8 manual confirm that our model is correct.

6 Formal Verification of Security Properties

In this section we prove an important security property, namely non-interference for the LEON3 processor.

6.1 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 [6] for details. We then show a single step theorem as below:

```
theorem single_step: "good_context s ⟹ NEXT s =
  Some (snd (fst (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 "sensible" or not.

6.2 Privilege Safety Theorem

Next we show that a successful one step execution in user mode does not lift the privilege to supervisor mode.

```
theorem privilege_safety:
assumes "get_delayed_pool s = [] ∧ get_trap_set s = {} ∧
  snd (execute_instruction() s) = False ∧
  s' = snd (fst (execute_instruction() s)) ∧
  ((ucast (get_S (cpu_reg_val PSR s)))::word1) = 0"
shows "((ucast (get_S (cpu_regval PSR s')))::word1) = 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 fourth conjunct says `s'` is the post-state, the last conjunct says the S bit in the pre-state `s` is 0 (i.e., `s` is in user mode). We show that the S bit in the post-state `s'` is also 0. This proof is a case analysis for each instruction and it checks that the execution mode is not modified.

6.3 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 [20]. 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 [26]. At the ISA level, this is similar to the non-infiltration property à la Khakpour et al. [20]. Here we first show that non-interference is preserved in single step executions.

```
theorem non_interference_step:
assumes "((ucast (get_S (cpu_reg_val PSR s1)))::word1) = 0
good_context s1 ∧ good_context s2 ∧ low_equal s1 s2 ∧
get_delayed_pool s1 = [] ∧ get_trap_set s1 = {} ∧
((ucast (get_S (cpu_reg_val PSR s2)))::word1) = 0 ∧
get_delayed_pool s2 = [] ∧ get_trap_set s2 = {}"
shows "∃ t1 t2. Some t1 = NEXT s1 ∧ Some t2 = NEXT s2 ∧
((ucast (get_S (cpu_reg_val PSR t1)))::word1) = 0 ∧
((ucast (get_S (cpu_reg_val PSR t2)))::word1) = 0 ∧
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:

```
low_equal s1 s2 ≡
(cpu_reg s1) = (cpu_reg s2) ∧ (user_reg s1) = (user_reg s2) ∧
(sys_reg s1) = (sys_reg s2) ∧ (∀ va. (virt_to_phys va (mmu s1)
(mem s1)) = (virt_to_phys va (mmu s2) (mem s2))) ∧
(∀ pa. (user_accessible s1 pa) → mem_equal s1 s2 pa) ∧
(mmu s1) = (mmu s2) ∧ (state_var s1) = (state_var s2) ∧
(traps s1) = (traps s2) ∧ (undef s1) = (undef s2)
```

Similarly 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_accessible` means that the physical address `pa` is accessible in state `s1`. Since we assume that `s1` and `s2` have the same MMU setup (including the virtual to physical address translation `virt_to_phys`), `pa` is also accessible in `s2`. `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 reminder 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 non_interference: assumes
"((ucast (get_S (cpu_reg_val PSR s1)))::word1) = 0 ∧
good_context s1 ∧ good_context s2 ∧ low_equal s1 s2 ∧
get_delayed_pool s1 = [] ∧ get_trap_set s1 = {} ∧
((ucast (get_S (cpu_reg_val PSR s2)))::word1) = 0 ∧
get_delayed_pool s2 = [] ∧ get_trap_set s2 = {} ∧
user_seq_exe n s1 ∧ user_seq_exe n s2"
shows "(∃ t1 t2. Some t1 = SEQ n s1 ∧ Some t2 = SEQ n s2 ∧
  ((ucast (get_S (cpu_reg_val PSR t1)))::word1) = 0 ∧
  ((ucast (get_S (cpu_reg_val PSR t2)))::word1) = 0 ∧
  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.

7 Conclusion

This paper describes the first formal model of the SPARCv8 ISA. Our formalisation has over 5000 lines of Isabelle code, not including the proofs. The model can be specialised to any SPARCv8 processor, and it contains many features specific to the SPARCv8 architecture. Our model is carefully designed to take advantage of the Isabelle code export functionality, through which we obtain executable code from our formal model.

We have validated our model against an official LEON3 simulator on more than 100k random instruction instances as well as real life programs. We believe our formalisation provides a solid foundation for future verification problems. 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.

With regard to machine code verification using our formal model, there are two possible angles in future work. First, although the provided operational semantics and the single step execution allow us to verify properties using Isabelle/HOL, automated reasoning about properties of machine code requires much work. Obtaining a functional representation of the SPARCv8 machine code in Isabelle/HOL from the semantics introduced in this work, in a similar fashion to [17], would ease the verification. Second, the memory model in our SPARCv8 formalisation is a strong consistency model, which is not suitable for verifying concurrent execution in modern day multi-core processors. This requires a weaker memory model, e.g., TSO, as well as a proof system for concurrency, such as Rely-Guarantee [19].

This research is supported in part by the National Research Foundation, Prime Minister's Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2014NCR-NCR001-30) and administered by the National Cybersecurity R&D Directorate.

References

1. ESA LEON processor. http://www.esa.int/Our_Activities/Space_Engineering_Technology/LEON_the_space_chip_that_Europe_built. [Online; accessed 27/01/2016].
2. K computer. http://www.top500.org/system/177232. [Online; accessed 27/01/2016].
3. L3 specification language for ISAs. http://www.cl.cam.ac.uk/~acjf3/l3/. [Online; accessed 09/12/2015].
4. LEON3 processor. http://www.gaisler.com/index.php/products/processors/leon3. [Online; accessed 27/10/2015].
5. RISC-V architecture. https://riscv.org/. [Online; accessed 10/08/2016].
6. Securify: Micro-kernel verification. http://securify.scse.ntu.edu.sg/MicroVer/. [Online; accessed 24/05/2016].
7. The SPARC architecture manual version 8. http://gaisler.com/doc/sparcv8.pdf. [Online; accessed 27/10/2015].
8. Tianhe-2. http://top500.org/system/177999. [Online; accessed 27/01/2016].
9. Xtratum hypervisor. http://www.xtratum.org/. [Online; accessed 29/01/2016].
10. R. Atkey. CoqJVM: An executable specification of the Java virtual machine using dependent types. In *TYPES*, LNCS, pages 18–32. Springer, 2005.
11. B. Campbell and I. Stark. Randomised testing of a microprocessor model using SMT-solver state generation. In *FMICS 2014*, pages 185–199. Springer, 2014.
12. 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.
13. 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.
14. A. Fox. Formal specification and verification of ARM6. In *Theorem Proving in Higher Order Logics*, volume 2758 of *LNCS*, pages 25–40. Springer, 2003.
15. A. Fox. Directions in ISA specification. In *Interactive Theorem Proving*, volume 7406 of *LNCS*, pages 338–344. Springer Berlin Heidelberg, 2012.
16. A. Fox. Improved tool support for machine-code decompilation in HOL4. In *Interactive Theorem Proving 2015*, pages 187–202, 2015.
17. A. Fox and M. O. Myreen. A trustworthy monadic formalization of the ARMv7 instruction set architecture. In *Interactive Theorem Proving*, pages 243–258, 2010.
18. S. Goel, W. A. Hunt, and M. Kaufmann. Abstract stobjs and their application to ISA modeling. In *ACL2 2013*, pages 54–69, 2013.
19. C. B. Jones. Specification and design of (parallel) programs. In *Proceedings of IFIP'83*, pages 321–332. North-Holland, 1983.
20. 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.
21. 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.

22. X. Leroy. The CompCert C verified compiler. `http://compcert.inria.fr/man/manual.pdf`, 2015. [Online; accessed 29/01/2016].
23. 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.
24. A. Santoro, W. Park, and D. Luckham. SPARC-V9 architecture specification with Rapide. Technical report, Stanford, CA, USA, 1995.
25. 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.
26. G. Smith. Principles of secure information flow analysis. In *Malware Detection*, pages 291–307, 2007.
27. 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.