Verifying Software Emulation of an Unsupported Hardware Instruction

Samuel Gruetter
MIT, USA

Thomas Bourgeat
EPFL, Switzerland

Adam Chlipala
MIT, USA

Abstract

Some processors, especially embedded ones, do not implement all instructions in hardware. Instead, if the processor encounters an unimplemented instruction, an unsupported-instruction exception is raised, and an exception handler is run that implements the missing instruction in software. Getting such a system to work correctly is tricky: The exception-handler code must not destroy any state of the user program and must use the control and status registers (CSRs) of the processor correctly. Moreover, parts of the handler are typically implemented in assembly, while other parts are implemented in a language like C, and one must make sure that when jumping from the user program into the handler assembly, from the handler assembly into C, back to assembly and finally back to the user program, all the assumptions made by the different pieces of code, hardware, and the compiler are satisfied.

Despite all these tricky details, there is a concise and intuitive way of stating the correctness of such a system: User programs running on a system where some instructions are implemented in software behave the same as if they were running on a system where all instructions are implemented in hardware.

We formalize and prove such a statement in the Coq proof assistant, for the case of a simple exception handler implementing the multiplication instruction on a RISC-V processor.

2012 ACM Subject Classification Theory of computation → Program reasoning; Theory of computation → Program verification; Theory of computation → Program specifications; Software and its engineering → Formal software verification

Keywords and phrases Software verification, Software-hardware boundary, Coq

Digital Object Identifier 10.4230/LIPIcs.ITP.2024.8

Supplementary Material *Software*: `https://github.com/mit-plv/softmul`

Funding This work was supported in part by National Science Foundation grants 1521584, 2130671, and 2217064, as well as a gift from Google.

1 Introduction

Assembly language is frequently regarded as the lowest level of software abstraction in software-verification endeavors. However, the ISA (instruction-set architecture) semantics typically employed for software verification present an abstraction of the bare-metal ISA specifications, omitting machine-level aspects of the ISA, like the configuration registers that control the intricate interplay between the hardware's intrinsic capabilities and the meticulously crafted firmware (a piece of software) tasked with maintaining machine configurations and implementing high-privilege handlers in charge of emulating unsupported instructions, as well as managing other forms of low-level exceptions.

For example, in the RISC-V ISA, control and status registers (CSRs) shape the behavior and functionality of the machine. These registers serve as a mechanism for controlling various

15th International Conference on Interactive Theorem Proving (ITP 2024).
Editors: Yves Bertot, Temur Kutsia, and Michael Norrish; Article No. 8; pp. 8:1–8:17
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

aspects of the processor's operation, ranging from enabling or disabling specific features to controlling where the machine jumps in case of interrupts and exceptions. These registers and the associated exception handlers exert fundamental control over machine behaviors, so their improper configuration can lead to undefined outcomes.

CSRs coupled with the handlers introduce an intriguing specification, implementation, and verification challenge: while they are essential to determining the machine's behavior, the CSRs are themselves set and manipulated by software, and the handlers are themselves software.

There is a bit of a chicken-and-egg problem: We want to provide a nice and simple ISA abstraction, but to implement this abstraction and prove it correct, we have to write a trap handler and want to compile parts of it with a compiler whose proof already relies on this abstraction that we are supposed to implement, so how can we break the circularity?

One might be tempted simply to augment software-verification efforts with more detailed and faithful ISA specifications. We eschew this approach. The simplified ISA abstractions commonly employed are far more practical and productive compared to their cumbersome and heavier bare-metal counterparts, and the intricate details of configurations and handlers should anyway remain irrelevant to software or compilers higher up the stack.

This paper endeavors to disentangle the problem by focusing on a *simplified-yet-illustrative* instance: the specification, implementation, and verification of a RISC-V machine with software-implemented multiply instructions.

Through this exploration, we aim to shed light on the interesting challenges posed by CSRs and handlers and pave the way for a more coherent understanding of hardware-software interactions.

We will show that for this simple case we can indeed provide (with proofs!) the desired abstractions, and we can leverage tools that were built on top of those nice abstractions to provide the said abstractions without creating a circular conundrum. Our solution is to prove a helper lemma that ports assembly program-correctness proofs against the nice and simple ISA semantics to proofs against the detailed low-level ISA semantics. The helper lemma requires that the program does not contain any unsupported instruction that would trigger the trap handler, and this assumption gets discharged when we instantiate it with the concrete handler code produced by the compiler. However, there are also parts of the handler whose semantics cannot be expressed using the nice and simple ISA semantics, and we implement these manually in assembly and prove their correctness directly at the assembly level.

Our paper makes the following contributions:

- We propose a pleasantly simple specification for a RISC-V system equipped with a software trap handler emulating unsupported instructions: User programs running on a system where some instructions are implemented in software in a trap handler should behave as if they were running on a system with hardware support for these instructions.
- We implement such a trap handler by combining code in a C-like language with handwritten assembly code, and we prove its correctness, in a mechanized and foundational way, down to the binary machine code of the handler, combining symbolic-evaluation proofs at the C level and assembly level with a compiler-correctness proof.

All our code is publicly available at `https://github.com/mit-plv/softmul`.

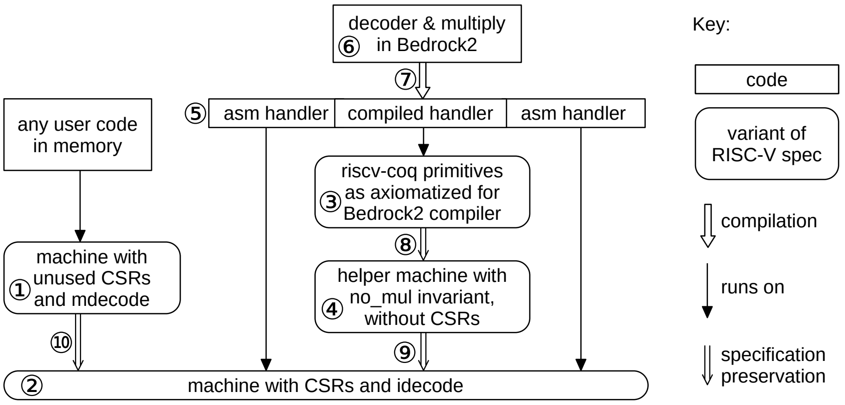
decoder & multiply in Bedrock2
⑥
⑦
⑤
asm handler
compiled handler
asm handler
any user code in memory
riscv-coq primitives as axiomatized for Bedrock2 compiler
③
⑧
machine with unused CSRs and mdecode
①
helper machine with no_mul invariant, without CSRs
④
⑩
⑨
②
machine with CSRs and idecode
Key:
code
variant of RISC-V spec
compilation
runs on
specification preservation

Figure 1 Overview diagram. The circled numbers are referenced in the text and do not stand for any meaningful order.

2 Overview

We want to show that a machine without hardware support for multiplication, but correctly configured with an exception handler that implements multiplication in software, behaves like a machine that supports multiplication in hardware. This theorem could then be used to simplify reasoning about programs running on a machine without hardware multiplication, because it saves the burden of reasoning about the trap handler and instead makes it as easy as reasoning about the specification with multiplication in hardware:

```
match inst with
| Mul rd rs1 rs2 ⇒
    x ← getRegister rs1;
    y ← getRegister rs2;
    setRegister rd (mul x y)
| ...
end
```

We use the RISC-V instruction-set architecture [1, 2], as formalized in riscv-coq [5]. RISC-V splits the instruction set into several extensions, each named with an uppercase letter. The base instruction set that every processor must support is called I, and multiplication, division and modulo operations are in a separate extension called M that small embedded processors may choose not to implement, or to implement in software by catching unsupported-instruction exceptions. In our proof-of-concept case study, we pretend that the M extension only contains one single instruction, namely the multiplication instruction, but we believe that support for the other instructions of the M extension could be added in the same way.

The riscv-coq specification defines a set of roughly a dozen *primitives* such as `getRegister`, `setRegister`, `loadByte`, `storeByte`, and then defines the semantics of each RISC-V instruction in terms of these primitives. As explained in [5], the semantics of each primitive is deliberately left unspecified in riscv-coq, so that each application that needs a formal specification of RISC-V can instantiate these primitives in a suitable domain-specific way.

Figure 1 presents an overview of our code (boxes ⑤ and ⑥) and specifications (the remaining boxes). Our theorem uses two instantiations of the riscv-coq specification: One that implements multiplication in hardware (box ①) and one (box ②) that implements it using a trap handler. Note that since the configurability of this specification is first-class, i.e.

expressed in Coq itself rather than in some configuration files of the build process, there is no code duplication between the two instantiations.

Parts of the exception handler (box ⑥) are implemented in the Bedrock2 source language [8] (a small and simple subset of C) and compiled (⑦) using the Bedrock2 compiler, but the handler also needs some low-level operations that are not expressible in the Bedrock2 source language and are therefore implemented by-hand in assembly. That is, our handler (box ⑤) starts and ends in handwritten assembly and calls a compiled Bedrock2 function in the middle. Our proof combines a program-logic proof about the Bedrock2 handler function, the compiler-correctness proof, and a proof about the assembly instructions, guaranteeing that all these parts have been put together correctly, and the final statement only mentions RISC-V semantics. All the other interfaces have been canceled out by combining the proofs and thus are not part of the trusted code base anymore.

In addition to the two instantiations of the RISC-V semantics with and without hardware multiplication, our proof (but not the final statement) also uses a third instantiation (box ④) which does not have any CSRs (control and status registers, required by the exception mechanism). This third instantiation fails (with undefined behavior) on all CSR-related instructions. For the compiler, an axiomatization (box ③) of this instantiation was chosen to simplify the proof, because the compiler does not emit any instructions that depend on CSRs.

3 The Top-Level Theorem Statement

We can state the theorem (arrow ⑩ in Figure 1) as follows:

```
Theorem softmul_correct: forall (initialH initialL: MachineState) (post: State → Prop),
  runsTo (mcomp_sat (run1 mdecode)) initialH post →
  R initialH initialL →
  runsTo (mcomp_sat (run1 idecode)) initialL (fun finalL ⇒
          exists finalH, R finalH finalL ∧ post finalH).
```

It is phrased as a *specification-preservation*[1] statement: If a machine with hardware multiplication runs from an initial state `initialH` to states satisfying a postcondition `post`, then every machine `initialL` with hardware multiplication, related to `initialH` by `R`, runs to a low-level state `finalL` which, when translated back to a high-level state `finalH`, satisfies the same postcondition.

The theorem uses `run1`, which defines how one single instruction is executed:

```
Definition run1(decoder: Z → Instruction): M unit :=
  pc ← getPC;
  inst ← Machine.loadWord Fetch pc;
  Execute.execute (decoder (LittleEndian.combine 4 inst));;
  endCycleNormal.
```

It is is parameterized over the instruction decoder, which is instantiated with `mdecode` (a decoder that supports the multiplication instruction) in the hypothesis and with `idecode` (a decoder that returns `InvalidInstruction` for the multiplication instruction) in the conclusion of the theorem.

The `mcomp_sat` function is of type `M unit → State → (State → Prop) → Prop` and asserts that a monadic program (consisting of primitives used in riscv-coq such as `getRegister`,

[1] It can also be seen as a *small-step omnisemantics forward simulation* as defined in [6].

```
Definition R(r1 r2: MachineState): Prop :=
  r1.(regs) = r2.(regs) ∧
  r1.(pc) = r2.(pc) ∧
  r1.(nextPc) = r2.(nextPc) ∧
  r1.(csrs) = map.empty ∧
  basic_CSRFields_supported r2 ∧
  regs_initialized r2.(regs) ∧
  exists mtvec_base scratch_end,
    map.get r2.(csrs) CSRField.MTVecBase = Some mtvec_base ∧
    map.get r2.(csrs) CSRField.MScratch = Some scratch_end ∧
    <{ * eq r1.(mem)
       * mem_available (word.of_Z (scratch_end - 256)) (word.of_Z scratch_end)
       * ptsto_bytes (word.of_Z (mtvec_base * 4)) softmul_binary }> r2.(mem).
```

Figure 2 The predicate relating high-level states (multiplication implemented in hardware) to low-level states (multiplication implemented in software)

`setRegister`, `loadByte`, etc.), applied to some initial state, satisfies a postcondition, and `runsTo` lifts it to an arbitrary (but finite) number of steps.[2] The predicate `R` (Figure 2) is used to relate a high-level state (i.e. the state of a machine that supports multiplication in hardware) to a low-level state (i.e. the state of a machine that implements multiplication in software using a trap handler), and it also contains all the preconditions on how the low-level machine needs to be configured. That is, `R` asserts that the two states have the same values for the registers and the program counter, and that the memory (modeled as a partial map from 32-bit addresses to bytes) of the low-level machine contains all of the high-level memory, as well as the instructions of the exception handler and some scratch space that the exception handler can use as its stack (which must be available even if the main program has used up all of its stack). To define at which address in memory the handler and the scratch space are located, RISC-V defines some CSRs [2] that our definition of `R` mentions:

- The CSR called `MTVecBase` is used to store the address of the trap handler (we use *direct* mode where all exceptions set the PC to the same address, but RISC-V also has a *vectored* mode where the PC is set to the base address in this register plus an offset corresponding to the cause of the exception).
- The CSR called `MScratch` is a read/write register dedicated for use by machine mode, and we use it to store the address of the *end* of the scratch space (we store the end address instead of the start address because it is used like a stack that grows downwards).

The memory (record fields `r1.(mem)` and `r2.(mem)` in Figure 2) is modeled as a finite map from 32-bit words to bytes. In the setup used in this case study, no primitive (nor other operation) changes the domain of that map. If an address outside of the domain of that map is accessed, the memory-access primitives cause undefined behavior, i.e. the `Prop` returned by `mcomp_sat` (and thus also the `Prop` returned by `runsTo`) becomes unprovable. This means that the `runsTo` hypothesis of the top-level theorem assumes a basic form of memory safety of the user program, namely that it does not access memory outside the domain of the memory. The separation-logic formula used in Figure 2 ensures that the memory the user program can write to (`r1.(mem)`) is disjoint from the scratch space and the handler code (second and third bullet points, respectively, in the separation-logic formula). To remove this memory-safety

[2] `runsTo` is defined like the omnisemantics *eventually* operator [6].

assumption, one could prove memory safety for the user program, i.e. that a `runsTo` holds for an arbitrary postcondition (the easiest choice would simply be λ`s. True`). In our setting, user code and handler code both run in machine mode, but in more complex systems that feature both user mode and machine mode and also hardware-based memory-protection support (e.g. by segmentation or virtual memory), the requirement to assume or prove this basic memory safety for user programs could be lifted.

4 The Handler Code

The exception-handler code is implemented partially in handwritten assembly and partially in the Bedrock2 [8] source language and compiled to bytes by the Bedrock2 compiler. In order to *prove* the `softmul_correct` theorem, we use the correctness theorem of the Bedrock2 compiler, but note that the *statement* of the `softmul_correct` theorem does not depend on the Bedrock2 language semantics or on anything related to the fact that we used the Bedrock2 compiler, so the auditing burden for someone (who trusts the Coq proof checker) auditing our handler is much smaller, because one does not need to worry about the compiler, its language semantics, and its interaction with the assembly code.

The handwritten assembly of the handler is shown in Figure 3a. Since we want our software-emulated multiplication to behave as if it were implemented in hardware, we cannot make any assumptions about the remaining space on the user program's stack, nor about whether the stack pointer `sp` contains any meaningful value at all. Therefore, we reserve a separate scratch space in memory just for our handler, and we require that the CSR `MScratch` contains the address of that scratch space.

As its first action (in `handler_init`), the handler has to store all 32 registers of the user process by which it was triggered. It may only use registers that it has already saved, because otherwise it would destroy state of the user program. We therefore resort to tricks such as temporarily storing the user stack pointer in the `MScratch` CSR and then temporarily storing it in the return-address register. Such tricks are easy to get wrong (and we did; see section 8.2).

After `handler_init`, the registers 3 to 31 are saved to the scratch space as well, and then the Bedrock2-generated part is called by passing it the value of the CSR register `MTVal`, which contains the invalid instruction that caused the exception, and a pointer to the scratch space in which we saved the registers.

The Bedrock2 code (Figure 3b) is written directly in Coq using the custom-notations feature, a C-like syntax, and operator precedence as suggested by whitespace. It extracts the three 5-bit fields of the instruction that indicate the two source registers (operands of the multiplication operation) and the destination register, respectively, and then calls another Bedrock2 function `rpmul` that implements multiplication in terms of addition, storing the result back into the scratch space. The `rpmul` function iterates over the bits of the second operand while repeatedly doubling the first operand, a technique sometimes called "Russian peasant multiplication." Both `softmul` and `rpmul` are verified using the Bedrock2 program logic. The spec of the former is given in Figure 4.

Its pre- and postcondition are expressed in terms of an (unused) I/O trace `t` and the memory `m`, for which we assert a list of two separation-logic clauses (a word array corresponding to the scratch space containing the register values, and a generic frame `R` for the rest of the memory).

After the Bedrock2 part, the handwritten snippet `inc_mepc` runs. It increases the CSR called `MEPC`, which stores the address of the instruction that caused the exception. This

```
Definition handler_init := [[
  Csrrw sp sp MScratch;  (* swap sp and MScratch CSR *)
  Sw sp zero (-128);     (* save the 0 register (for uniformity) *)
  Sw sp ra (-124);       (* save ra *)
  Csrr ra MScratch;      (* use ra as a temporary register... *)
  Sw sp ra (-120);       (* ... to save the original sp *)
  Csrw sp MScratch;      (* restore the original value of MScratch *)
  Addi sp sp (-128)      (* remainder of code will be relative to updated sp *)
]].

Definition call_mul := [[
  Csrr a0 MTVal;  (* argument 0: value of invalid instruction *)
  Addi a1 sp 0;   (* argument 1: pointer to memory with register values before trap *)
  Jal ra (Z.of_nat (1 + List.length inc_mepc + 29 + List.length handler_final) * 4)
]].

Definition inc_mepc := [[
  Csrr t1 MEPC;
  Addi t1 t1 4;
  Csrw t1 MEPC
]].

Definition handler_final := [[
  Lw ra sp 4;
  Lw sp sp 8; (* Bug: used to be `Csrr sp MScratch`, which is wrong if Mul sets sp *)
  Mret
]].

Definition asm_handler_insts := handler_init ++ save_regs3to31 ++
  call_mul ++ inc_mepc ++ restore_regs3to31 ++ handler_final.
```

(a) Assembly part of trap handler (embedded in Coq)

```
Definition softmul := func! (inst, a_regs) {
  a = a_regs + (inst>>15 & 31)<<2;
  b = a_regs + (inst>>20 & 31)<<2;
  d = a_regs + (inst>>07 & 31)<<2;
  unpack! c = rpmul(load(a), load(b));
  store(d, c)
}.

Definition rpmul := func! (x, e) ⤳ ret {
  ret = $0;
  while (e) {
    if (e & $1) { ret = ret + x };
    e = e >> $1;
    x = x + x
  }
}.
```

(b) Bedrock2 part of trap handler (using custom Coq notations to make it look similar to C)

Figure 3 Trap handler code

```
Instance spec_of_softmul : spec_of "softmul" :=
  fnspec! "softmul" inst a_regs / rd rs1 rs2 regvals R,
  { requires t m :=
      mdecode (word.unsigned inst) = MInstruction (Mul rd rs1 rs2) ∧
      List.length regvals = 32 ∧
      seps [a_regs ↦ word_array regvals; R] m;
    ensures t' m' := t = t' ∧
      seps [a_regs ↦ word_array (List.upd regvals (Z.to_nat rd) (word.mul
               (List.nth (Z.to_nat rs1) regvals default)
               (List.nth (Z.to_nat rs2) regvals default))); R] m' }.
```

Figure 4 Specification of softmul function

increment is needed because upon returning from the trap handler (by the `Mret` instruction), execution will jump to `MEPC`, so we have to set it to one instruction (i.e., 4 bytes) past the multiplication instruction.

And finally, in `restore_regs3to31` and `handler_final`, the values of the user program's registers are restored.

5 Combining the Program-Logic Proofs and Compiler-Correctness Proof

By combining the program-logic proofs about the two Bedrock2 functions with the compiler-correctness theorem, we can prove that if we run the compiler within Coq to obtain a list of instructions `mul_insts`, these instructions satisfy the specification shown in Figure 5, a verbose but unsurprising specification, laying out calling-convention details.

Lines 5 to 6 specify in which registers the arguments need to be placed, and line 14 requires that at address `a_regs`, there is an array of 32 words that store the values of the registers of the user program. Lines 18 to 20 state that after running `mul_insts`, the array at address `a_regs` storing the registers is updated at its `rd`'th index with the result of multiplying its `rs1`-th and `rs2`-th elements, and line 23 states that the new registers of the processor (not the ones saved in memory) only differ from the original registers on the callee-saved registers.

Note that the conclusion on line 27 refers to the same machine as the conclusion of the top-level theorem in section 3, namely the one described by `(mcomp_sat (run1 idecode))`, or box ② in Figure 1. However, to get there, two more proof steps (⑧ and ⑨) are needed: In order to keep the Bedrock2 compiler (somewhat) general, it was not proven against a specific instantiation of the riscv-coq semantics but against an axiomatization (box ③) of the primitives used in riscv-coq such as `getRegister`, `setRegister`, `loadByte`, etc. However, to keep the Bedrock2 compiler proof manageable, the RISC-V machine-state representation appearing in that axiomatization was hardcoded to a record type without CSRs (because compiler-emitted code never touches CSRs).

An additional problem requiring some proof effort to show compatibility is that the compiler correctness proof assumes a machine with hardware support for multiplication, but we want to run its code on one without. By inspecting the code that it generated, we can see that it did not output any multiplication instructions, but if it did, this would lead to a serious bug: If during the execution of the trap handler, a multiplication instruction were encountered, the trap handler would be recursively invoked again, infinitely many times.

We solve these two problems by introducing an intermediate helper machine (box ④)

```
Lemma mul_correct: forall initial a_regs regvals invalidIInst R (post: State → Prop)
                          ret_addr stack_start stack_pastend rd rs1 rs2,
   word.unsigned initial.(pc) mod 4 = 0 →
   initial.(nextPc) = word.add initial.(pc) (word.of_Z 4) →
   map.get initial.(regs) RegisterNames.a0 = Some invalidIInst →
   map.get initial.(regs) RegisterNames.a1 = Some a_regs →
   map.get initial.(regs) RegisterNames.ra = Some ret_addr →
   map.get initial.(regs) RegisterNames.sp = Some stack_pastend →
   word.unsigned ret_addr mod 4 = 0 →
   word.unsigned (word.sub stack_pastend stack_start) mod 4 = 0 →
   regs_initialized initial.(regs) →
   mdecode (word.unsigned invalidIInst) = MInstruction (Mul rd rs1 rs2) →
   128 ≤ word.unsigned (word.sub stack_pastend stack_start) →
   seps [a_regs ↦ with_len 32 word_array regvals;
         initial.(pc) ↦ program idecode mul_insts;
         mem_available stack_start stack_pastend; R] initial.(MinimalCSRs.mem) ∧
   (forall newMem newRegs,
     seps [a_regs ↦ with_len 32 word_array (List.upd regvals (Z.to_nat rd) (word.mul
                (List.nth (Z.to_nat rs1) regvals default)
                (List.nth (Z.to_nat rs2) regvals default)));
           initial.(pc) ↦ program idecode mul_insts;
           mem_available stack_start stack_pastend; R] newMem →
     map.only_differ initial.(regs) reg_class.caller_saved newRegs →
     regs_initialized newRegs →
     post { initial with pc := ret_addr; nextPc := word.add ret_addr (word.of_Z 4);
             MinimalCSRs.mem := newMem; regs := newRegs }) →
   runsTo (mcomp_sat (run1 idecode)) initial post.
```

Figure 5 The correctness lemma of the compiler-generated part of the handler

that uses the same state representation (without CSRs) as the compiler, and we prove an invariant `no_mul` saying that the memory region marked as executable (which only includes the compiled handler code in that instance) contains no multiplication instructions.

6 Correctness Proof of the Assembly Part

The assembly part of the handler is proven correct by induction over the `runsTo` hypothesis of `softmul_correct`. If the machine with hardware multiplication executes any instruction besides multiplication, we just need to show that after executing the same instruction on the machine with software multiplication, the `R` judgment is preserved, but we can do that once-and-for-all by inspecting each *primitive* of the riscv-coq spec (`getRegister`, `setRegister`, `loadByte`, etc.), instead of analyzing the much larger number of RISC-V *instructions*. The interesting case is when the machine with hardware multiplication encounters a multiplication instruction, and we have to show that the machine with software multiplication steps to a related state. We do so by first symbolically executing the specification of what the *hardware* does in case of an exception (Figure 6), which boils down to setting some CSR fields and then setting the PC to the exception-handler address found in the `MTVecBase` CSR.

After that, we symbolically execute the handwritten assembly instructions, using Coq's proof context to keep track of all the facts that we know about the current state of the machine. For each assembly instruction, we encounter its specification in terms of the primitives of riscv-coq, and for each primitive, we have a helper lemma that updates our symbolic state. At the point where we reach the call to the Bedrock2-generated code, we apply the correctness lemma for the compiled trap handler. After that call, we step through more handwritten assembly instructions that restore the registers and then call the `Mret` instruction that jumps back to one instruction past the multiplication instruction that caused

```
Definition raiseExceptionWithInfo{A: Type}(isInterrupt exceptionCode info: t): M A :=
  pc ← getPC;
  (* hardcoded simplification: we only support machine mode and no interrupts *)
  addr ← getCSRField MTVecBase;
  setCSRField MTVal (regToZ_unsigned info);;
  (* these two need to be set just so that Mret will succeed at restoring them *)
  setCSRField MPP (encodePrivMode Machine);;
  setCSRField MPIE 0;;
  setCSRField MEPC (regToZ_unsigned pc);;
  setCSRField MCauseCode (regToZ_unsigned exceptionCode);;
  setPC (ZToReg (addr * 4));;
  @endCycleEarly M t MM MW MP A.
```

Figure 6 Specification (in riscv-coq) of what hardware does in case of an exception

the exception. At that point, we need to prove that the symbolic state accumulated in the Coq proof context implies that the two machines are still related by `R`, which only works if there are no bugs in the handler code.

7 What If . . .

To explain our specification from a different angle, we list a few potential bugs that an implementor could introduce, and we show how they make our specification unprovable. Note that these are not bugs that actually occurred in our own implementation. For those, we refer to section 8.2. To present each potential bug, we ask: What if . . .

- . . . the compiler used to compile the handler emitted a multiplication instruction, which would cause the handler to trigger itself recursively infinitely many times? When proving correctness of the handwritten assembly (section 6), when we get to the jump instruction that calls the code emitted by the Bedrock2 compiler, we need to apply the compiler-correctness theorem (instantiated with the Bedrock2 part of our handler), but that theorem talks about execution on a machine with multiplication support, whereas the theorem we are about to prove is about execution on a machine without multiplication support. To make the proof work, we need to introduce box ④ and steps ⑧ and ⑨ in Figure 1 as explained in section 5, which at some point requires us to go through the concrete list of instructions emitted by the compiler and to check that none of them is a multiplication instruction.
- . . . the handler runs at a time when no stack exists or the stack does not have enough remaining space? The output of the Bedrock2 compiler contains a number that indicates the amount of stack space that the compiled code needs, and one hypothesis of the compiler-correctness theorem is that at least that much space is available below the current stack pointer. In order to make sure this hypothesis holds, our trap handler uses a separate reserved scratch pad in memory as its stack, and when the correctness theorem for the handwritten assembly applies the instantiated compiler-correctness theorem `mul_correct`, it has to prove that there are at least 128 bytes of space remaining in the scratch pad, as mandated by the hypothesis on line 13 in Figure 5.
- . . . the assembly that calls compiled Bedrock2 code makes wrong assumptions about the calling conventions of the compiler, e.g. which registers are used to pass arguments, or whether they are passed on the stack, in which direction the stack grows, or which registers are caller-saved? All these conventions are also captured in the intermediate

lemma `mul_correct` in Figure 5.

- . . . the handler forgot to increase MEPC, the CSR storing the address to which the machine jumps when we return from the exception handler, which would cause the faulting multiplication instruction to be run again and trigger the handler again? At the end of the handler correctness proof, this bug would lead to a mismatch between the state of the machine with multiplication support (whose program counter gets advanced past the multiplication instruction) and the state of the machine without multiplication support (whose program counter would still point to the multiplication instruction).
- . . . we ran a user program using compressed instructions (2-byte instructions) on our system? The riscv-coq specification only supports the uncompressed instruction format, where all instructions are 4 bytes long. There is no single location where the spec explicitly says "compressed instructions are *not* supported" – it requires an attentive reader who notices that the whole spec never mentions compressed instructions. In this scenario, our trap handler would fail to decode the unsupported instruction, and arbitrary behavior would occur. If riscv-coq did support compressed instructions, and our handler correctly decoded them, that would still require it to decide correctly whether to increase the MEPC by 2 or 4, and like in the previous point, one would notice the mismatch during the proof.

8 Evaluation

We attempt to answer the following evaluation questions (and dedicate one subsection to each of them):

1. Does our verified trap handler run on a RISC-V system implemented by a third party?
2. Did our implementation contain bugs that our verification caught?
3. Did our implementation contain bugs that our verification failed to catch?
4. Was the effort required for verification lower than the effort for debugging would have been?

8.1 Running Our Handler

To validate that our verified handler actually runs on a system not implemented by ourselves, we first looked for small embedded RISC-V processors without multiplication support but could not find any product with enough documentation in English to make us want to try it out. Instead, we chose to test our code in the Spike RISC-V ISA simulator [3], which offers fine-grained control over which RISC-V extensions are enabled.

We want to test that our handler behaves as expected on a system that runs a simple C program with multiplications, compiled by a third-party compiler. We wrote a simple program which computes the factorial of a hardcoded number and saves the result as well as a "done" flag to memory. We compiled it using the GNU RISC-V toolchain.

Our top-level theorem applies to a list of bytes called `softmul_binary` (mentioned in Figure 2 in the definition of the relation `R`), representing a piece of position-independent RISC-V machine code. However, Spike expects as input an ELF file. We relied on the GNU RISC-V toolchain to transform our binary into an ELF file, using a custom 25-line linker script.

For our theorem to be applicable, the conditions that the relation `R` (Figure 2) imposes on `r2` (the machine without support for multiplication) must hold on our Spike machine. The first six conditions above the **`exists`** are related to the formalization and do not require any special setup action. The two lines below the **`exists`** require that the `MTVecBase` and `MScratch`

CSRs have suitable values, which we ensure by running an assembly script at the beginning that initializes these two CSRs with addresses defined in our linker script. The last three lines are a bullet-point separation-logic clause list describing the memory, saying that it must contain all of the specification machine's memory `r1.(mem)`, as well as 256 bytes of scratch memory at the address in the `MScratch` CSR and the `softmul_binary` at the address in the `MTVecBase` CSR. Our linker script, together with the memory-layout command-line argument we pass to Spike, ensures that these conditions hold.

Spike comes with its own small language of debugger commands, and we used it to run the system until the done flag in memory is 1, then print the value of the memory at the address where we expect the result, and we also print the value of the CSR `minstret`, the number of retired instructions, to see how many instructions were executed.

No matter whether we invoked Spike with or without multiplication enabled, we observed the same result for `factorial(5)`, namely 120. With multiplication enabled, the number of instructions was 87; and with multiplication disabled, the number of instructions increased to 787, which shows that our handler indeed ran. As an additional sanity check, we also confirmed that it stops working if we set the `MTVecBase` CSR to a different value.

Therefore, at least for this one simple example, we can answer question 1 with 'yes.'

8.2 Bugs Caught During Verification

At the end of the proof that steps through the handwritten handler assembly, we need to prove that the symbolic state accumulated in the Coq proof context implies that the two machines are still related by `R`, which only works if there are no bugs in the handler code (see end of section 6). At that point, we found two interesting bugs. The first one was that we forgot to reset the `MScratch` CSR, so one invocation of the exception handler works fine, but the next one will use a wrong address for its scratch space. The second bug was the corner case where the multiplication instruction stores its result into the stack pointer. In that case, we must not override the stack pointer with the original stack pointer that we swapped into the `MScratch` register at the beginning of the handler.

We also found two more obvious bugs related to when to set the stack pointer and what stack-pointer offsets to use.

So we can answer question 2 with 'yes.'

8.3 Bugs Encountered While Trying to Run It

We split the development of our experiment into two phases: First, we set up the linker script, with the trap handler already in place, but inactive, because we enabled the M extension. Once this experiment produced the expected output, we deactivated the M extension, so that our handler would run.

Getting phase 1 to work required some debugging. The most difficult part was to understand how to pass the linker-script-defined address of the heap memory to the C program, and it required reading the relevant page[3] of the GNU Linker's manual, which starts by saying that "accessing a linker script defined variable from source code is not intuitive," and further down explains that "when you are using a linker script defined symbol in source code you should always take the address of the symbol, and never attempt to use its value".

[3] https://sourceware.org/binutils/docs/ld/Source-Code-Reference.html

None of the code involved in phase 1 was verified, so it is not surprising that debugging was required. And to our delight, in phase 2, as soon as we disabled the M extension, our verified trap handler worked on the first try, and no debugging was needed at all.

So, to answer question 3, there were bugs in the unverified part, but no bugs in the verified part.

In the future, it would be interesting also to verify ELF file generation, which we believe could have prevented the above bug.

8.4 Effort

For lack of better measures, we resort to lines-of-code counts as a very approximate measure of effort. Table 1 lists the counts of the different components.

It suggests that to produce 76 lines of verified code, a total of 3331 lines of code was necessary, which is more than a 40× blowup. This ratio looks not very appealing, but it still seems fair to say that for tricky code, large proofs are sometimes needed. We also have some (potentially alleviating) remarks for each row of the table:

- The RISC-V helper instance is not referenced by the top-level theorem statement but acts as a bridge between the RISC-V spec used by the Bedrock2 compiler (whose state does not contain any CSRs) and the one used in the top-level theorem (whose state does have CSRs). Additionally, the helper instance maintains the invariant that no executable instructions are from the M extension, which is important during the execution of the trap handler, because if the trap handler contained a multiplication instruction, the trap handler would be invoked recursively over and over again. The helper instance and its accompanying lemmas are mostly copied from the one used in the compiler, and careful refactoring to share the code with the compiler could considerably reduce this count, which also means that these lines were low-effort to produce.
- To verify multiplication and a simple instruction decoder in Bedrock2, we used the original Bedrock2 program logic [8], which only automates the application of weakest-precondition rules but does not provide any automation for side condition solving. Using a framework that provides more automation would have reduced this proof size.
- A large chunk of the proof lines (1454) is in the correctness proof of the trap-handler parts written in assembly. The reason for this verbosity might be that, to our knowledge, this project is the first within the Bedrock2 ecosystem to verify more than two or three lines of assembly at a time, so there was no assembly-specific framework available. About two thirds of the proof code could probably be factored out into a framework that would be reusable for other assembly programs as well. We also did not spend too much time on side-condition automation, which could further reduce the number of proof lines. We conjecture that in a more mature assembly-verification framework, the assembly part of the trap-handler proof might be as short as maybe 100 lines of code. Moreover, the code-to-proof ratio also looks bad because we count the number of lines of Coq code rather than the number of assembly instructions, which matters for `save_regs3to31` and `restore_regs3to31`: Each of these is just a two-line functional program but expands to 29 assembly instructions.
- The compiler compat & invocation code deals with the different RISC-V instances and decoders and also applies the Bedrock2 compiler's correctness theorem for the instruction decoder and multiplier implemented in Bedrock2. It consists of important but not particularly interesting bookkeeping that quickly adds up to many lines of proof.
- Finally, the top-level theorem puts everything together. It requires some helper lemmas that could probably be generalized and moved to a library, but the fact that these lemmas

	impl	spec	proof	total
RISC-V helper instance	0	101	309	410
Multiplication in Bedrock2	8	5	83	96
Instruction decoder in Bedrock2	7	27	80	114
Trap handler in assembly	36	28	1454	1518
Compiler compat & invocation	14	47	716	777
Top-level theorem	11	18	147	176
Excluded (imports & comments)				240
Total	76	226	2789	3331

Table 1 Lines-of-code counts, excluding the dependencies (coqutil, riscv-coq, Bedrock2, and the Bedrock2 compiler)

were not already present in any library used in the Bedrock2 ecosystem seems fairly representative of the general verification experience, so it seems fair to count these lines.

Finding the bugs described in section 8.2 through debugging (especially the first two) might have been quite hard but would probably still not have taken as long as our verification effort took, so the answer for question 4 is probably a ‘no.’

However, we can imagine a promising world where the proof burden becomes lower than the debugging burden and verification becomes a part of most systems developers’ toolboxes.

9 Related Work

A number of projects have attempted to verify the interaction between (some or all of) C code, its compilation, handwritten assembly code, and trap handlers.

In the context of the Verisoft project, Alkassar et al. [4] verified a virtual-memory system that can swap out virtual memory pages onto disk. If an address is accessed that currently is on disk, a page fault is triggered, and a verified page-fault handler runs. Their correctness statement says that a physical machine with the page-fault handler can simulate a virtual machine (by which they mean a machine that provides to a user process a linear memory covering the whole address space). Their handler is implemented in C0 (a subset of C) with some inline assembly, which is modeled as external calls that modify additional state that cannot be modified directly from C0. That is, they call assembly from C, whereas we chose the opposite direction, calling C (or the C-like language Bedrock2, in our case) from assembly. In their project, saving and restoring of registers before and after the handler are not implemented in assembly and verified like we do but are instead part of the semantics of the physical machine.

BabyVMM [16] proves correctness of a simple virtual memory manager by showing that for all kernel implementations, linking the kernel with the virtual memory manager and running it on a machine with only physical memory (“hardware model” HW) behaves like running the kernel on a machine with an address space whose lower part is physical memory and whose upper part is virtual memory (“address space model” AS). It is implemented in a C-like language, and no compiler nor assembly code appears in the formalization. Instead, the theorem is stated in terms of C semantics. It also does not mention any page-fault handlers.

The verified microkernel seL4 [12] is implemented in C, but some small parts are handwritten assembly and are not verified [14, sections 4.4 and 4.8]. Contrary to our approach of using a verified compiler, they apply translation validation to the binary generated by GCC and certify using SMT solvers that it behaves like the C program.

CertiKOS [10, 11, 7] is a verified OS kernel. By means of certified abstraction layers, it fully captures the behavior of each component in a deep specification, so that from the outside, it does not matter whether the component is implemented in C or in assembly, thus achieving interoperability at the proof level between C and assembly. Its correctness is expressed as a contextual refinement, based on CompCert's [15] notion of a backward simulation, extended with a universal quantification over all possible surrounding programs (contexts): It states that for all assembly programs, all behaviors of that assembly program when linked with the low-level kernel can be simulated by the same program when linked with the high-level kernel specification. It relies on a notion of linking and uses CompCert's formalization of assembly, which is still fairly high-level compared to binary machine code, e.g. jumps use labels instead of offsets or addresses, and there are instructions that allocate and free a stack frame that do not correspond to any machine instructions. CompCert's assembly (which is used to model CertiKOS's lowest layer) also does not model CSRs, whereas riscv-coq, on which our project is based, does, so to model trap handlers at our level of detail, the assembly (or machine) model would have to be extended.

CompCertELF [17], a different project by the same group, extends CompCert to also cover machine-code generation and uses a more realistic memory model, without the stack-frame allocation/freeing instructions mentioned above. As far as we know, CompCertELF has not (yet) been integrated with CertiKOS and is not publicly available. If it were, and if we managed to make CompCertELF compatible with our project, it could have helped to prevent the bug (section 8.3) we encountered in our unverified usage of the GNU linker to turn our plain binary into an ELF file.

Goel et al. [9] verify a subset of the instructions of an x86 processor which decodes x86 instructions and translates them into micro-operations before executing them. For the more complex instructions, the generated micro-operations contain a trap that causes a jump to microcode stored in a ROM. Similarly to our theorem, they prove that this processor behaves as if there were no micro-operations, traps or microcode, and instructions were executed according to a high-level x86 specification.

The CakeML compiler [13] targets multiple ISAs, and some instructions (e.g. division) are not supported by all of them, so the compiler has to implement some unsupported instructions in software, but contrary to our work, the necessary in-software implementation is emitted directly by the compiler, and no trap handler comes into play.

10 Conclusion and Future Work

We have shown a pleasantly simple way of specifying the correctness of a trap handler that emulates unsupported instructions in software, and we proved that our implementation of such a trap handler combining handwritten assembly and compiler-generated code satisfies this specification by combining symbolic-evaluation proofs about assembly and Bedrock2 programs with the correctness proof of the Bedrock2 compiler, as well as by proving that the output of the Bedrock2 compiler, which assumes a machine without CSRs and with hardware support for multiplication, also runs correctly on a machine with CSRs but without hardware support for multiplication.

This style of proof relating multiple execution models constitutes a first step towards the more ambitious goal of thoroughly proving correctness of a virtual memory system, stated in a similar flavor by saying that user programs running on a system with virtual memory (implemented by a combination of hardware, assembly, and C) behave as if they were running on a machine where the user program can use the full physical address space.

References

1 The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213. *RISC-V Foundation*, December 2019. URL: https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf.

2 The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 20211203. *RISC-V International*, December 2021. URL: https://github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf.

3 Spike RISC-V ISA Simulator, July 2023. URL: https://github.com/riscv-software-src/riscv-isa-sim.

4 Eyad Alkassar, Norbert Schirmer, and Artem Starostin. Formal Pervasive Verification of a Paging Mechanism. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 109–123, Berlin, Heidelberg, 2008. Springer. doi:10.1007/978-3-540-78800-3_9.

5 Thomas Bourgeat, Ian Clester, Andres Erbsen, Samuel Gruetter, Pratap Singh, Andy Wright, and Adam Chlipala. Flexible Instruction-Set Semantics via Abstract Monads (Experience Report). *Proceedings of the ACM on Programming Languages*, 7(ICFP):192:108–192:124, August 2023. URL: https://dl.acm.org/doi/10.1145/3607833, doi:10.1145/3607833.

6 Arthur Charguéraud, Adam Chlipala, Andres Erbsen, and Samuel Gruetter. Omnisemantics: Smooth Handling of Nondeterminism. *ACM Transactions on Programming Languages and Systems*, 45(1):5:1–5:43, March 2023. URL: https://dl.acm.org/doi/10.1145/3579834, doi:10.1145/3579834.

7 Hao Chen, Xiongnan Wu, Zhong Shao, Joshua Lockerman, and Ronghui Gu. Toward Compositional Verification of Interruptible OS Kernels and Device Drivers. *Journal of Automated Reasoning*, 61(1):141–189, June 2018. doi:10.1007/s10817-017-9446-0.

8 Andres Erbsen, Samuel Gruetter, Joonwon Choi, Clark Wood, and Adam Chlipala. Integration verification across software and hardware for a simple embedded system. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, pages 604–619, New York, NY, USA, June 2021. Association for Computing Machinery. doi:10.1145/3453483.3454065.

9 Shilpi Goel, Anna Slobodova, Rob Sumners, and Sol Swords. Verifying x86 instruction implementations. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, pages 47–60, New York, NY, USA, January 2020. Association for Computing Machinery. doi:10.1145/3372885.3373811.

10 Ronghui Gu, Jeremie Koenig, Tahina Ramananandro, Zhong Shao, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. Deep Specifications and Certified Abstraction Layers. Technical Report Technical Report YALEU/DCS/TR-1500, Yale University, October 2014.

11 Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 653–669, USA, November 2016. USENIX Association.

12 Gernot Heiser. The seL4 Microkernel – An Introduction. Technical report, June 2020. URL: https://sel4.systems/About/seL4-whitepaper.pdf.

13 Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. The verified CakeML compiler backend. *Journal of Functional Programming*, 29:e2, 2019. URL: https://www.cambridge.org/core/product/identifier/S0956796818000229/type/journal_article, doi:10.1017/S0956796818000229.

14 Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *ACM*

Transactions on Computer Systems, 32(1):2:1–2:70, February 2014. URL: https://dl.acm.org/doi/10.1145/2560537, doi:10.1145/2560537.

15 Xavier Leroy. Formal Verification of a Realistic Compiler. *Commun. ACM*, 52(7):107–115, July 2009. doi:10.1145/1538788.1538814.

16 Alexander Vaynberg and Zhong Shao. Compositional Verification of a Baby Virtual Memory Manager. In Chris Hawblitzel and Dale Miller, editors, *Certified Programs and Proofs*, pages 143–159, Berlin, Heidelberg, 2012. Springer. doi:10.1007/978-3-642-35308-6_13.

17 Yuting Wang, Xiangzhe Xu, Pierre Wilke, and Zhong Shao. CompCertELF: Verified separate compilation of C programs into ELF object files. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):197:1–197:28, November 2020. URL: https://dl.acm.org/doi/10.1145/3428265, doi:10.1145/3428265.