Proving that a System with Software Trap Handlers for Unimplemented Instructions Behaves as if They Were Implemented in Hardware

SAMUEL GRUETTER, THOMAS BOURGEAT, ADAM CHLIPALA, MIT CSAIL, USA

Some processors, especially embedded ones, do not implement all instructions in hardware. Instead, if execution encounters an unimplemented instruction, an unsupported-instruction exception is raised, and an exception handler is run which 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 comply with all restrictions imposed by the processor. 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 all the assumptions made by the assembly code, the C code, and the compiler are satisfied, even though the code is executed in a context where commonly valid assumptions might not hold.

But 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 show how to formalize and prove this statement in the Coq proof assistant, for the case of a simple exception handler implementing the multiplication instruction on a RISC-V processor.

1 INTRODUCTION

Our theorem uses two instantiations of the riscv-coq specification [Bourgeat et al. 2022]: One that implements multiplication in hardware, and one 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.

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
```

Parts of the exception handler are implemented in the Bedrock2 source language [Erbsen et al. 2021] and compiled 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. 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 any more.

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 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, this

instantiation was chosen to simplify the proof, because the compiler does not emit any instructions that depend on CSRs.

2 THE THEOREM STATEMENT

We can state the theorem as follows:

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

It uses `run1`, a function that defines how one single instruction is executed, which is parameterized over the instruction decoder, and to which we pass `mdecode` (a decoder that supports the multiplication instruction) in the hypothesis and `idecode` (a decoder that returns `InvalidInstruction` for the multiplication instruction) in the conclusion:

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

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`, `setRegister`, `loadByte`, etc), applied to some initial state, satisfies a postcondition, and `runsTo` lifts it to an arbitrary (but finite) number of steps. The predicate `related` (Figure 1) 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, `related` 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 everything 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 control-and-status registers (CSRs) that our definition of `related` 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 this scratch space (we store the end address instead of the start address because it is used like a stack that grows downwards).

So overall, the theorem `softmul_correct` can be read as follows: If a machine with hardware multiplication runs to a high-level state satisfying a postcondition, then every related machine with software multiplication runs to a low-level state which, when translated back to a high-level state, satisfies the same postcondition.

```
Definition related(r1 r2: State): 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 stacktrash stack_hi,
    map.get r2.(csrs) CSRField.MTVecBase = Some mtvec_base ∧
    map.get r2.(csrs) CSRField.MScratch = Some stack_hi ∧
    List.length stacktrash = 32%nat ∧
    seps [eq r1.(mem);
          word.of_Z (stack_hi - 128) ↦ word_array stacktrash;
          LowerPipeline.mem_available (word.of_Z (stack_hi - 256))
                                      (word.of_Z (stack_hi - 128));
          word.of_Z (mtvec_base * 4) ↦ program idecode handler_insts] r2.(mem).
```

Fig. 1. The predicate relating high-level states (multiplication implemented in hardware) to low-level states (multiplication implemented in software)

3 THE HANDLER CODE

The exception-handler code is implemented partially in handwritten assembly and partially in the Bedrock2 [Erbsen et al. 2021] 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 first few instructions of our handler (handwritten in Coq) are as follows:

```
Definition handler_init :=
[[  Csrrw sp sp MScratch;    (* swap stack pointer (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 *)
```

After that, 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 is written directly in Coq using the custom-notations feature, a C-like syntax, and operator precedence as suggested by whitespace in this example:

```
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
  }
}.
```

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:

```
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' }.
```

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).

4 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 obtain that if we run the compiler within Coq to obtain a list of instructions `mul_insts`, these instructions satisfy the specification shown in Figure 2.

5 CORRECTNESS PROOF OF 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 `related` 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 *instructions* that RISC-V has. The interesting case is when the machine with hardware multiplication encounters a multiplication instruction, and we have

```
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;
          LowerPipeline.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;
            LowerPipeline.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.
```

Fig. 2. The correctness lemma of the compiler-generated part of the handler

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, which boils down to setting some CSR fields and then setting the PC to the exception-handler address found in the `MTVecBase` CSR:

```
Definition raiseExceptionWithInfo{A: Type}(isInterrupt exceptionCode info: t): M A :=
  pc ← getPC;
  (* here we hardcode that this simplified spec only supports 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.
```

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 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`, which only works if there are no bugs in the handler code.

6 BUGS FOUND DURING VERIFICATION

At that final point in the proof described above, we actually 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.

Finding these bugs through debugging (especially the first two) might have been quite hard, so even though with today's state of the art of theorem proving, the proofs probably took even longer than the debugging, we can imagine a promising world where the proof burden becomes lower than the debugging burden and verification becomes a part of most systems developer's toolboxes.

REFERENCES

Thomas Bourgeat, Ian Clester, Andres Erbsen, Samuel Gruetter, Pratap Singh, Andrew Wright, and Adam Chlipala. 2022. Flexible Instruction-Set Semantics via Type Classes. https://doi.org/10.48550/arXiv.2104.00762 arXiv:arXiv:2104.00762

Andres Erbsen, Samuel Gruetter, Joonwon Choi, Clark Wood, and Adam Chlipala. 2021. Integration Verification Across Software and Hardware for a Simple Embedded System. *PLDI'21* (2021). https://doi.org/10.1145/3453483.3454065