Formal Methods for Hardware-Software Integration on RISC-V Embedded Systems

Samuel Gruetter

Joint work with Joonwon Choi, Andres Erbsen, Clark Wood, Adam Chlipala

RISC-V Summit, 11 Dec 2019
Rethink the way ISAs are done
Rethink the way ISAs are done stacks

<table>
<thead>
<tr>
<th>System Behavior Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applications</td>
</tr>
<tr>
<td>Library API Specs</td>
</tr>
<tr>
<td>Libraries</td>
</tr>
<tr>
<td>Programming Language Spec</td>
</tr>
<tr>
<td>Compiler</td>
</tr>
<tr>
<td>RISC-V ISA</td>
</tr>
<tr>
<td>Processor</td>
</tr>
<tr>
<td>Hardware Description Language Spec</td>
</tr>
</tbody>
</table>
Need agreement at each layer boundary

- System Behavior Requirements
- Applications
- Library API Specs
- Libraries
- Programming Language Spec
- Compiler
- RISC-V ISA
- Processor
- Hardware Description Language Spec
Interface specs take many forms

The RISC-V Instruction Set Manual
Volume I: User-Level ISA
Document Version 2.2
## Interface specs take many forms

<table>
<thead>
<tr>
<th>Address Range</th>
<th>Memory Attribute</th>
<th>Description</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1001_3000</td>
<td>RW</td>
<td>UART 0</td>
<td>On-Chip Peripherals</td>
</tr>
<tr>
<td>0x1001_4000</td>
<td>RW</td>
<td>QSPI0 Control</td>
<td></td>
</tr>
<tr>
<td>0x1001_5000</td>
<td>RW</td>
<td>PWM 0</td>
<td></td>
</tr>
<tr>
<td>0x1001_6000</td>
<td>RW</td>
<td><strong>Reserved</strong></td>
<td></td>
</tr>
<tr>
<td>0x1002_3000</td>
<td>RW</td>
<td>UART 1</td>
<td></td>
</tr>
<tr>
<td>0x1002_4000</td>
<td>RW</td>
<td>QSPI 1</td>
<td></td>
</tr>
<tr>
<td>0x1002_5000</td>
<td>RW</td>
<td>PWM 1</td>
<td></td>
</tr>
<tr>
<td>0x1002_6000</td>
<td>RW</td>
<td><strong>Reserved</strong></td>
<td></td>
</tr>
<tr>
<td>0x1003_4000</td>
<td>RW</td>
<td>QSPI 2</td>
<td>Off-Chip Non-Volatile Memory</td>
</tr>
<tr>
<td>0x1003_5000</td>
<td>RW</td>
<td>PWM 2</td>
<td></td>
</tr>
<tr>
<td>0x1003_6000</td>
<td>RW</td>
<td><strong>Reserved</strong></td>
<td></td>
</tr>
<tr>
<td>0x2000_0000</td>
<td>RXC</td>
<td>QSPI 0 XIP (512MiB)</td>
<td></td>
</tr>
<tr>
<td>0x4000_0000</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x8000_0000</td>
<td>RXWC</td>
<td>Data Tightly Integrated Memory (DTIM) 16 KiB</td>
<td>On-Chip Volatile Memory</td>
</tr>
<tr>
<td>0x8000_4000</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.1: FE310-G000 Memory Map.  
Memory Attributes: **R** - Read, **W** - Write, **X** - Execute, **C** - Cacheable
Interface specs take many forms

```java
public class TreeSet<E>
extends AbstractSet<E>
implements NavigableSet<E>, Cloneable, Serializable

A NavigableSet implementation based on a TreeMap. The elements are ordered using their natural ordering, or by a Comparator provided at set creation time, depending on which constructor is used.

This implementation provides guaranteed log(n) time cost for the basic operations (add, remove and contains).

Note that the ordering maintained by a set (whether or not an explicit comparator is provided) must be consistent with equals if it is to correctly implement the Set interface. (See Comparable or Comparator for a precise definition of consistent with equals.) This is so because the Set interface is defined in terms of the equals operation, but a TreeSet instance performs all element comparisons using its compareTo (or compare) method, so two elements that are deemed equal by this method are, from the standpoint of the set, equal. The behavior of a set is well-defined even if its ordering is inconsistent with equals; it just fail to obey the general contract of the Set interface.
```
Interface specs take many forms
Interface specs take many forms
Interface specs take many forms
Different levels of rigidity

- Some interface specs are (implicitly...) updated every day

- Others have stabilized and stay mostly the same over time
Problem

No matter

• how the specs are stated
• how often they change
• whether providers and users are the same team or not
Problem

No matter
- how the specs are stated
- how often they change
- whether providers and users are the same team or not

There will always be
- Misunderstandings
- Oversights
Problem

No matter
• how the specs are stated
• how often they change
• whether providers and users are the same team or not

There will always be
• Misunderstandings
• Oversights

Which lead to
• Bugs
• Security vulnerabilities
Problem

No matter
- how the specs are stated
- how often they change
- whether providers and users are the same team or not

There will always be
- Misunderstandings
- Oversights
Which lead to
- Bugs
- Security vulnerabilities

Can a computer help us solve this problem?

That’s our research
Prototype: The IoT lightbulb

- Wrote all digital parts from scratch
- Using bedrock2
  - a low-level programming language with verification support
- Very simple at the moment
- Goal: Should scale to more realistic applications
printf '1' | nc -w0 -u 192.168.1.123 9999
printf '0' | nc -w0 -u 192.168.1.123 9999
Specification using a Log

Recv1
💡 On
PollNone
Recv0
💡 Off
RecvInvalid
PollNone
PollNone
The lightbulb spec

`lightbulb_spec`:

The log of the blue wires and the red wire respects the following regular expression:

```
BootSeq ((Recv1 💡On) | (Recv0 💡Off) | RecvInvalid | PollNone)*
```
Verifying that our system satisfies lightbulb_spec

Implementation of a system

Specification (desired property)

Proof instructions

The Coq Proof Assistant

IDE

YES

NO
The end-to-end theorem

Using Coq, we developed

- a software image (list of bytes)
- a pipelined processor
- a theorem:

If you put the image at address 0 and set PC to 0 and run our processor, the log of the blue wires and the red wire respects the following regular expression:

\[
\text{BootSeq } ((\text{Recv1 } \text{💡On}) \mid (\text{Recv0 } \text{💡Off}) \mid \text{RecvInvalid} \mid \text{PollNone})^*)
\]
The end-to-end theorem provides:

- a concise description of the behavior of the whole stack
- high assurance of correctness

<table>
<thead>
<tr>
<th>System Behavior Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applications</td>
</tr>
<tr>
<td>Library API Specs</td>
</tr>
<tr>
<td>Libraries</td>
</tr>
<tr>
<td>Programming Language Spec</td>
</tr>
<tr>
<td>Compiler</td>
</tr>
<tr>
<td>RISC-V ISA</td>
</tr>
<tr>
<td>Processor</td>
</tr>
<tr>
<td>Hardware Description Language Spec</td>
</tr>
</tbody>
</table>
Systems development using machine-checked specs at all levels

- Applications
- Library API Specs
- Libraries
- Programming Language Spec
- Compiler
- RISC-V ISA
- Processor
- Hardware Description Language Spec
Systems development using machine-checked specs at all levels

- System Behavior Requirements
- Applications
- Library API Specs
- Libraries
- Programming Language Spec
- Compiler
- RISC-V ISA
- Processor
- Hardware Description Language Spec
Systems development using machine-checked specs at all levels

- System Behavior Requirements
- Applications
- Library API Specs
- Libraries
- Programming Language Spec
- Compiler
- RISC-V ISA
- Processor
- Hardware Description Language Spec

![Diagram of system development process with arrows indicating dependencies and checks.]
Invariant Inv:
- PC mod 4 = 0
- p\_insts ≤ PC ≤ p\_insts + L
- Memory looks as follows:

<table>
<thead>
<tr>
<th>p_insts</th>
<th>only ADD, SUB, MUL instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>p_insts + L</td>
<td>JAL $r0 -L</td>
</tr>
</tbody>
</table>
Invariant Inv:
- PC mod 4 = 0
- p\_insts ≤ PC ≤ p\_insts + L

- Memory looks as follows:

<table>
<thead>
<tr>
<th>p_insts</th>
<th>p_insts + L</th>
</tr>
</thead>
<tbody>
<tr>
<td>only ADD, SUB, MUL instructions</td>
<td>JAL $r0 -L</td>
</tr>
</tbody>
</table>

**Theorem:** If an RV32IM machine satisfies Inv, then it still satisfies Inv after running one instruction.

**Proof:**
- Case 1) p\_insts ≤ PC < p\_insts + L
  - Case 1.1) PC points to an ADD
    Spec says how state is updated, PC := PC + 4
  - Case 1.2) PC points to an SUB
    ...
  - Case 1.3) PC points to a MUL
    ...
- Case 2) PC = p\_insts + L
  Therefore PC points to JAL $r0 -L
  Spec says PC := PC + (-L) = p\_insts + L - L = p\_insts
Invariant Inv:
- PC mod 4 = 0
- $p_{\text{insts}} \leq PC \leq p_{\text{insts}} + L$
- Memory looks as follows:

<table>
<thead>
<tr>
<th>$p_{\text{insts}}$</th>
<th>only ADD, SUB, MUL instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_{\text{insts}} + L$</td>
<td>JAL $r0 - L$</td>
</tr>
</tbody>
</table>

**Theorem:** If an RV32IM machine satisfies Inv, then no matter for how many cycles we run it, it will always satisfy Inv.

**Proof:** follows from previous theorem (by induction on number of cycles).
**Compiler correctness proof**

### Invariant:
- PC remains within bounds
- Instruction memory unchanged
- Stack memory modified according to rules
- Data memory modified like source program P does
- I/O like source program P

<table>
<thead>
<tr>
<th>p_insts</th>
<th>instructions emitted by compiler for given program P</th>
</tr>
</thead>
<tbody>
<tr>
<td>p_insts + L</td>
<td>Memory</td>
</tr>
</tbody>
</table>
The stack

- System Behavior Requirements
- Applications
- Library API Specs
- Libraries
- Programming Language Spec
- Compiler
- RISC-V ISA
- Processor
- Hardware Description Language Spec
The stack

- Applications
- Library API Specs
- Libraries
- Programming Language Spec
- Compiler
- RISC-V ISA
- Processor
- Hardware Description Language Spec
DIVW and DIVUW instructions are RV64I instructions, and divide the lower 32 bits of \( rs1 \) by the lower 32 bits of \( rs2 \), treating them as signed and unsigned integers respectively, placing the 32-bit quotient in \( rd \), sign-extended to 64 bits. REMW and REMUW instructions are only valid for RV64, and provide the corresponding signed and unsigned remainder operations respectively. Both REMW and REMUW always sign-extend the 32-bit result to 64 bits, including on a divide by zero.

The semantics for division by zero and division overflow are summarized in Table 6.1. The quotient of division by zero has all bits set, and the remainder of division by zero equals the dividend. Signed division overflow occurs only when the most-negative integer is divided by \(-1\). The quotient of a signed division with overflow is equal to the dividend, and the remainder is zero. Unsigned division overflow cannot occur.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Division by zero</td>
<td>( x )</td>
<td>0</td>
<td>( 2^L - 1 )</td>
<td>( x )</td>
<td>( -1 )</td>
<td>( x )</td>
</tr>
<tr>
<td>Overflow (signed only)</td>
<td>(-2^{L-1})</td>
<td>(-1)</td>
<td>( - )</td>
<td>( - )</td>
<td>(-2^{L-1})</td>
<td>( 0 )</td>
</tr>
</tbody>
</table>

Table 6.1: Semantics for division by zero and division overflow. \( L \) is the width of the operation in bits: XLEN for DIV[U] and REM[U], or 32 for DIV[U]W and REM[U]W.

We considered raising exceptions on integer divide by zero, with these exceptions causing a trap in most execution environments. However, this would be the only arithmetic trap in the standard ISA (floating-point exceptions set flags and write default values, but do not cause traps) and would require language implementors to interact with the execution environment’s trap handlers.
DIVW and DIVUW instructions divide the lower 32 bits of rs2, treating the 32-bit quotient in rd, sign-extended for RV64, and provide the correct result. Both REMW and REMUW always divide by zero.

The semantics for division by zero has all bits set. Division overflow occurs only when signed division with overflow is enabled, and overflow cannot occur.

<table>
<thead>
<tr>
<th>Condition</th>
<th>Division</th>
<th>Overflow</th>
</tr>
</thead>
<tbody>
<tr>
<td>Division by zero</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Overflow (signed only)</td>
<td>$-2^{31}$</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6.1: Semantics for division by zero
Platform Specifics

The execution environment determines the mapping of a hart’s address space into memory, or provides dedicated I/O devices. Different execution environments on the same platform can cause some portions of main memory to be invisible to certain harts. Portions of memory can be mapped into the same or different address ranges for different platform-specific execution environments.

When a RISC-V platform has multiple harts, the address ranges of the harts can be the same, or entirely different, or may be partly differently mapped into the same or different address ranges. Different execution environments can affect portions of main memory.

Table 3.1: FE310-G000 Memory Map.
Memory Attributes: R - Read W - Write X - Execute C - Cache

<table>
<thead>
<tr>
<th>Address</th>
<th>Mask</th>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1002_5000</td>
<td>0x1002_5FFF</td>
<td>RW</td>
<td>PWM 1</td>
</tr>
<tr>
<td>0x1002_6000</td>
<td>0x1003_3FFF</td>
<td>RW</td>
<td>QSPI 2</td>
</tr>
<tr>
<td>0x1003_4000</td>
<td>0x1003_4FFF</td>
<td>RW</td>
<td>PWM 2</td>
</tr>
<tr>
<td>0x1003_5000</td>
<td>0x1003_5FFF</td>
<td>RW</td>
<td>Reserved</td>
</tr>
<tr>
<td>0x1003_6000</td>
<td>0x1FFF_FFFF</td>
<td>RXC</td>
<td>QSPI 0 XIP (512 MiB)</td>
</tr>
<tr>
<td>0x2000_0000</td>
<td>0x3FFF_FFFF</td>
<td>RW</td>
<td>Data Tightly Integrated Mem (DTIM) 16 KiB</td>
</tr>
<tr>
<td>0x4000_0000</td>
<td>0x7FFF_FFFF</td>
<td>RWXC</td>
<td>Reserved</td>
</tr>
<tr>
<td>0x8000_0000</td>
<td>0x8000_3FFF</td>
<td>RWXC</td>
<td>Data Tightly Integrated Mem (DTIM) 16 KiB</td>
</tr>
<tr>
<td>0x8000_4000</td>
<td>0x8FFF_FFFF</td>
<td>RWXC</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

RISC-V spec
Spec by chip vendor
Platform Specifics

```c
addr <- translate Load (ZToReg 1) (add a (x <- loadByte Execute addr; setRegister rd (int8ToReg x))
| Lh rd rs1 oimm12 =>
a <- getRegister rs1;
addr <- translate Load (ZToReg 2) (add a (x <- loadHalf Execute addr; setRegister rd (int16ToReg x))
| Lw rd rs1 oimm12 =>
a <- getRegister rs1;
addr <- translate Load (ZToReg 4) (add a (x <- loadWord Execute addr; setRegister rd (int32ToReg x))
| Lbu rd rs1 oimm12 =>
```

RISC-V spec
Platform Specifics

RISC-V spec

Spec by chip vendor
Parametrized stack
Can’t we just do extensive testing instead?

Our method can exhaustively test

✔ For all possible parameters

✔ For all possible inputs

\[
v_1 \quad v_2 \quad \ldots
\]

\[
2^{32} \times 2^{32} \times \ldots
\]

✔ For all numbers of cycles the system runs

1, 2, 3, ... ∞
All proofs in continuous integration
Why isn’t everyone doing this?
*The length of these bars is just my perception, we do not (yet) have any empirical values.*
Hasn’t this been done before?

Yes, but we’re not satisfied
Hasn’t this been done before?

The CLI/Piton stack (1980s-90s, in nqthm)
- BigInt multiplication
- Gates

CompCert C compiler (recent, in Coq)
- High-level specs
- VST
- Subset of C
- CompCert
- Idealized assembly

CakeML ML compiler (recent, in HOL4)
- ML
- Silver ISA
- Verilog
Our project (bedrock2)

- Powerful language for interface specs
- Parametrized stack
- Can swap our processor with a commercial RISC-V processor
- Actually communicates with the external world
  - Rather than just writing some result into the memory
Conclusion

RISC-V
- Simple
- Open
- Extensible

Formal Methods
- Enable machine and human readable specs
- Can do exhaustive verification of highly parametrized systems

Great match to take software and hardware development to the next level
Questions?

https://github.com/mit-plv/bedrock2
https://github.com/mit-plv/riscv-coq

✉️ gruetter@mit.edu  🌐️ LinkedIn  🍄 Twitter  @samuelgruetter
Additional slides
| Mul rd rs1 rs2 =>
  x <- getRegister rs1; y <- getRegister rs2; setRegister rd (mul x y)
| Mulh rd rs1 rs2 =>
  x <- getRegister rs1;
  y <- getRegister rs2;
  setRegister rd (highBits (regToZ_signed x * regToZ_signed y))
| Mulhsu rd rs1 rs2 =>
  x <- getRegister rs1;
  y <- getRegister rs2;
  setRegister rd (highBits (regToZ_signed x * regToZ_unsigned y))
| Mulhu rd rs1 rs2 =>
  x <- getRegister rs1;
  y <- getRegister rs2;
  setRegister rd (highBits (regToZ_unsigned x * regToZ_unsigned y))
| Div rd rs1 rs2 =>
  x <- getRegister rs1;
  y <- getRegister rs2;
  setRegister rd
| Mul rd rs1 rs2 =>
  x <- getRegister rs1; y <- getRegister rs2; setRegister rd (mul x y)
| Mulh rd rs1 rs2 =>
  x <- getRegister rs1;
  y <- getRegister rs2;
  setRegister rd (highBits (regToZ_signed x * regToZ_signed y))
| Mulhsu rd rs1 rs2 =>
  x <- getRegister rs1;
  y <- getRegister rs2;
  setRegister rd (highBits (regToZ_signed x * regToZ_unsigned y))
| Muhlu rd rs1 rs2 =>
  x <- getRegister rs1;
  y <- getRegister rs2;
  setRegister rd (highBits (regToZ_unsigned x * regToZ_unsigned y))
| Div rd rs1 rs2 =>
  x <- getRegister rs1;
  y <- getRegister rs2;
  setRegister rd

unfold highBits in *; simpl in *.
Coq provides precise commands for symbolic manipulation of specs
The spec is more than just a program

RISC-V spec

addr <- translate Load (ZToReg 1) (add a (x <- loadByte Execute addr; setRegister rd (int8ToReg x))
| Lh rd rs1 oimm12 =>
a <- getRegister rs1;
addr <- translate Load (ZToReg 2) (x <- loadHalf Execute addr; setRegister rd (int16ToReg x))
| Lw rd rs1 oimm12 =>
a <- getRegister rs1;
addr <- translate Load (ZToReg 4) (add a (x <- \textcolor{red}{LoadWord} Execute addr; setRegister rd (int32ToReg x))
| Lbu rd rs1 oimm12 =>

 Might perform memory-mapped I/O (MMIO). Load could return any 32bit value.
The spec is more than just a program

RISC-V spec

Can assign all possible 32bit integers at once

Might perform memory-mapped I/O (MMIO). Load could return any 32bit value.
Unverified glue code
Unverified glue code
Unverified glue code
Unverified glue code

"look similar"

"only trivial code"
Unverified glue code

Glue code

Glue code

Glue code
Unverified glue code