Semantics for Verified Software-Hardware Stacks

CS 152

Samuel Gruetter

Reusing/citing slides by Nada Amin
Based on joint work with Joonwon Choi, Andres Erbsen, Clark Wood, Adam Chlipala

Harvard, 21 April 2022

CSAIL
MIT COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE LABORATORY

Massachusetts Institute of Technology

About me

- Undergrad/masters at EPFL (Switzerland), supervised by then-PhD-student Nada Amin
- Internship at Princeton University with Andrew Appel's group using the Verfied Software Toolchain (VST)
- Currently: PhD student at MIT in Adam Chlipala's group

Styles of Semantics

Operational Semantics
Denotational Semantics
Axiomatic Semantics
Algebraic Semantics

↑

Techniques from CS152
Our PLDI'21 paper

↓

Integration Verification Across Software and Hardware for a Simple Embedded System

Andres Erbsen* Samuel Gruetter* Joonwon Choi Clark Wood Adam Chlipala
MIT CSAIL

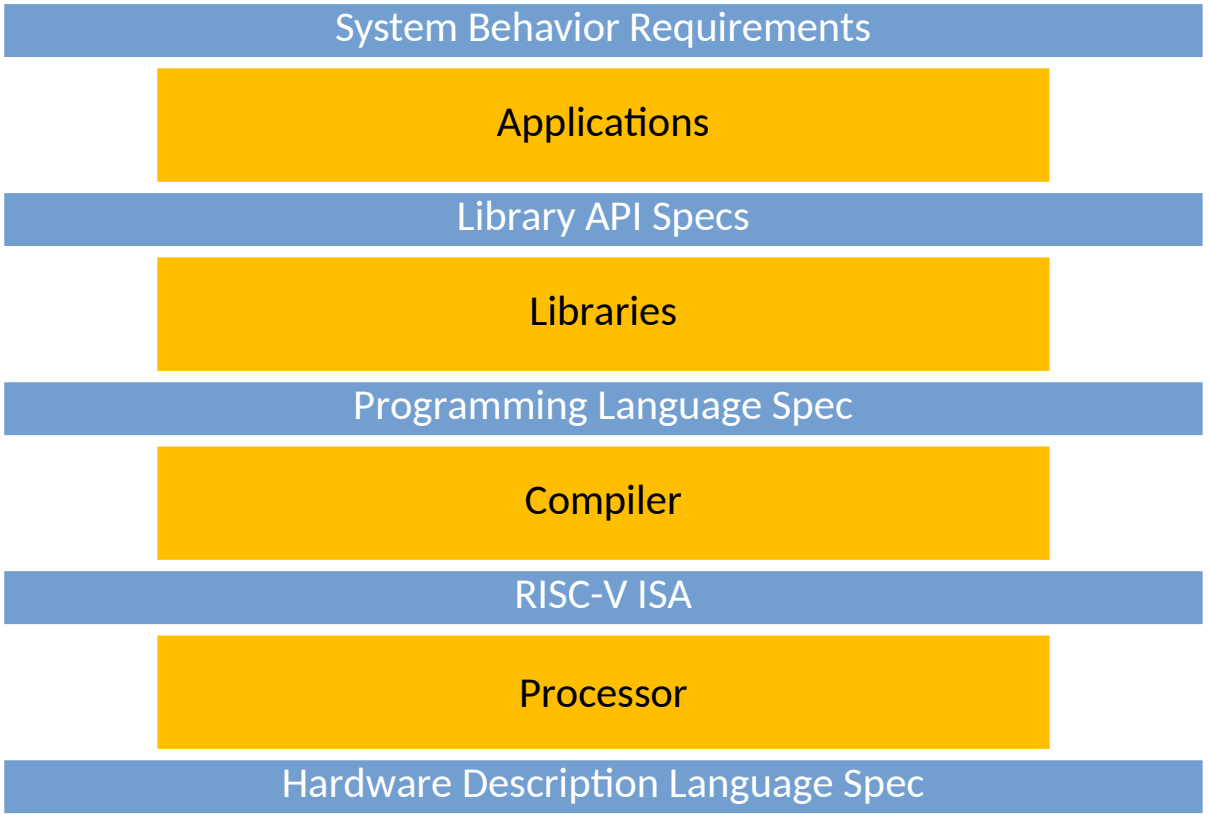
Abstract

The interfaces between layers of a system are susceptible to bugs if developers of adjacent layers proceed under subtly different assumptions. Formal verification of two layers

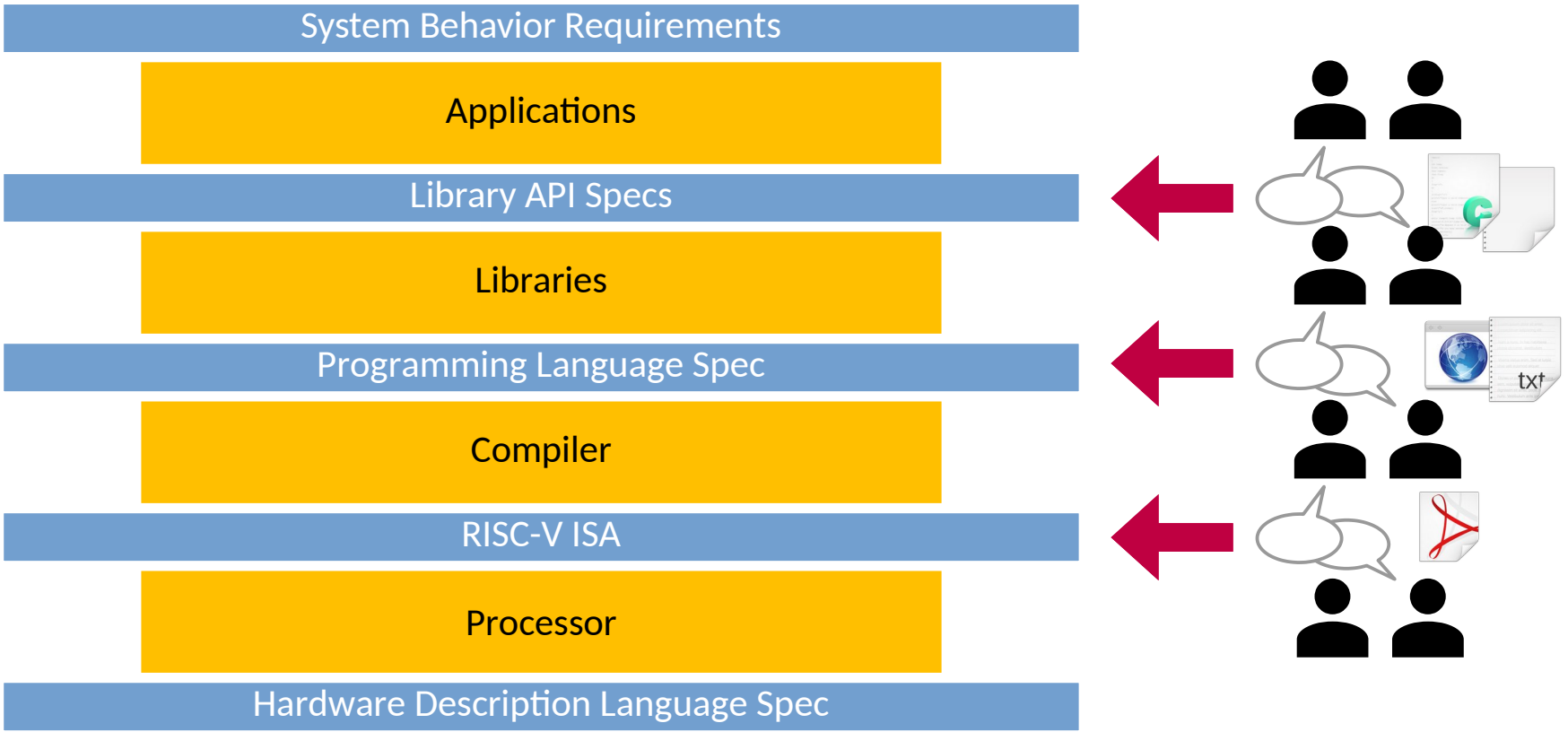
1 Introduction

We present the comprehensive and modular verification of functional correctness of a newly realistic but still very simple embedded system, highlighting important challenges

Software-Hardware Stacks



Need agreement at each layer boundary



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

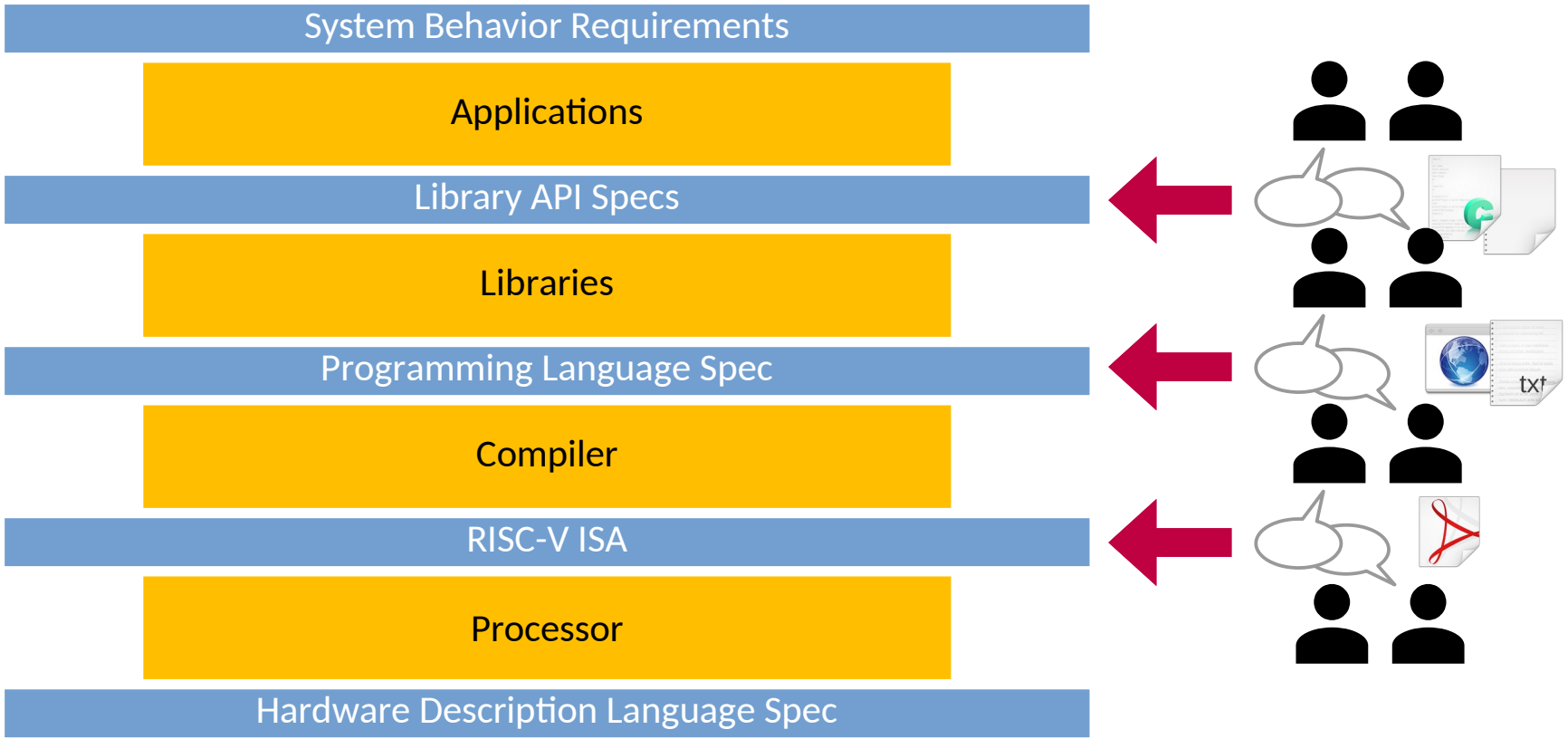
No
- or not

There will always be
- Bugs
- Security vulnerabilities

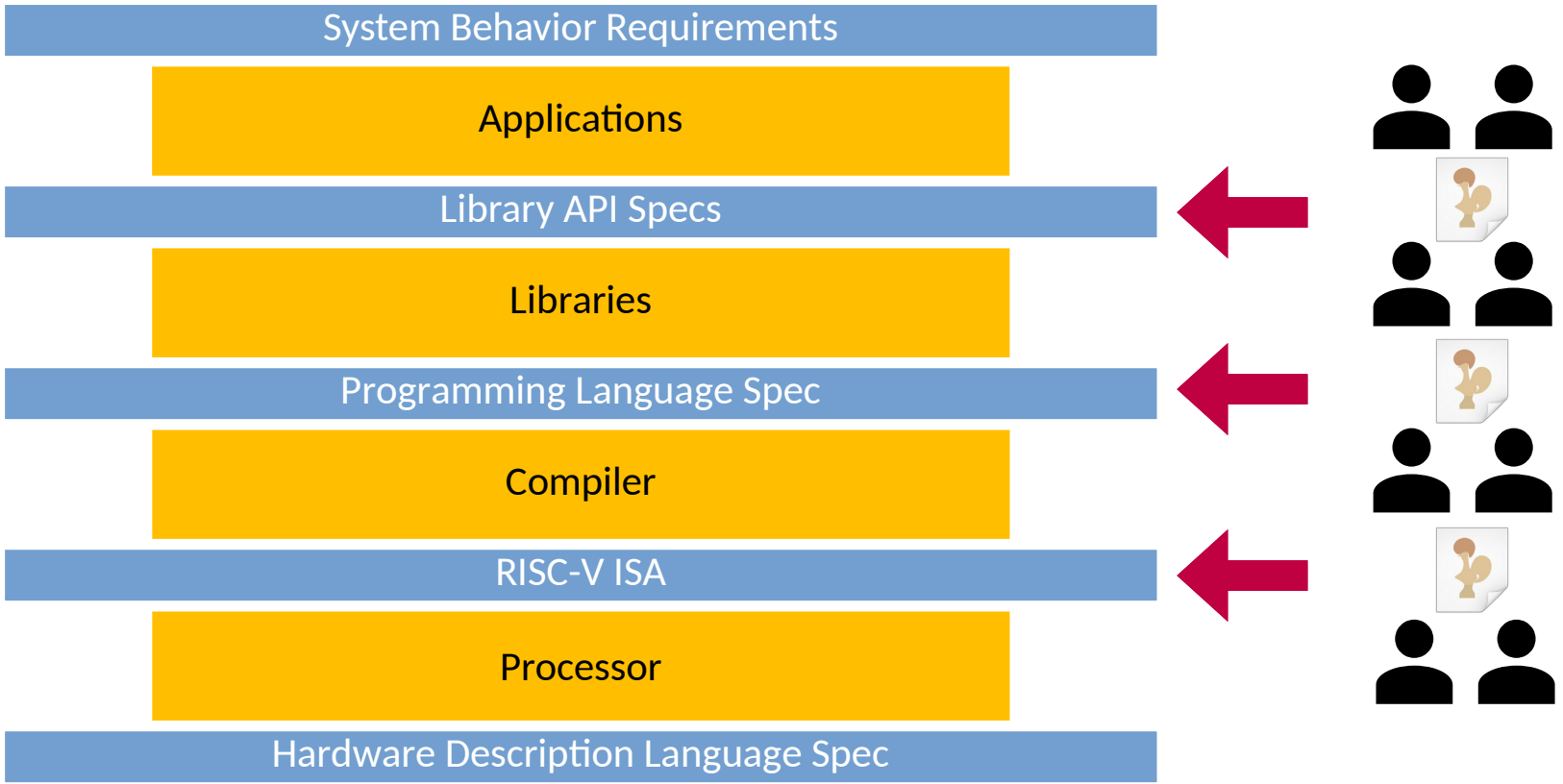
Can a computer help us solve this problem?

That’s our research

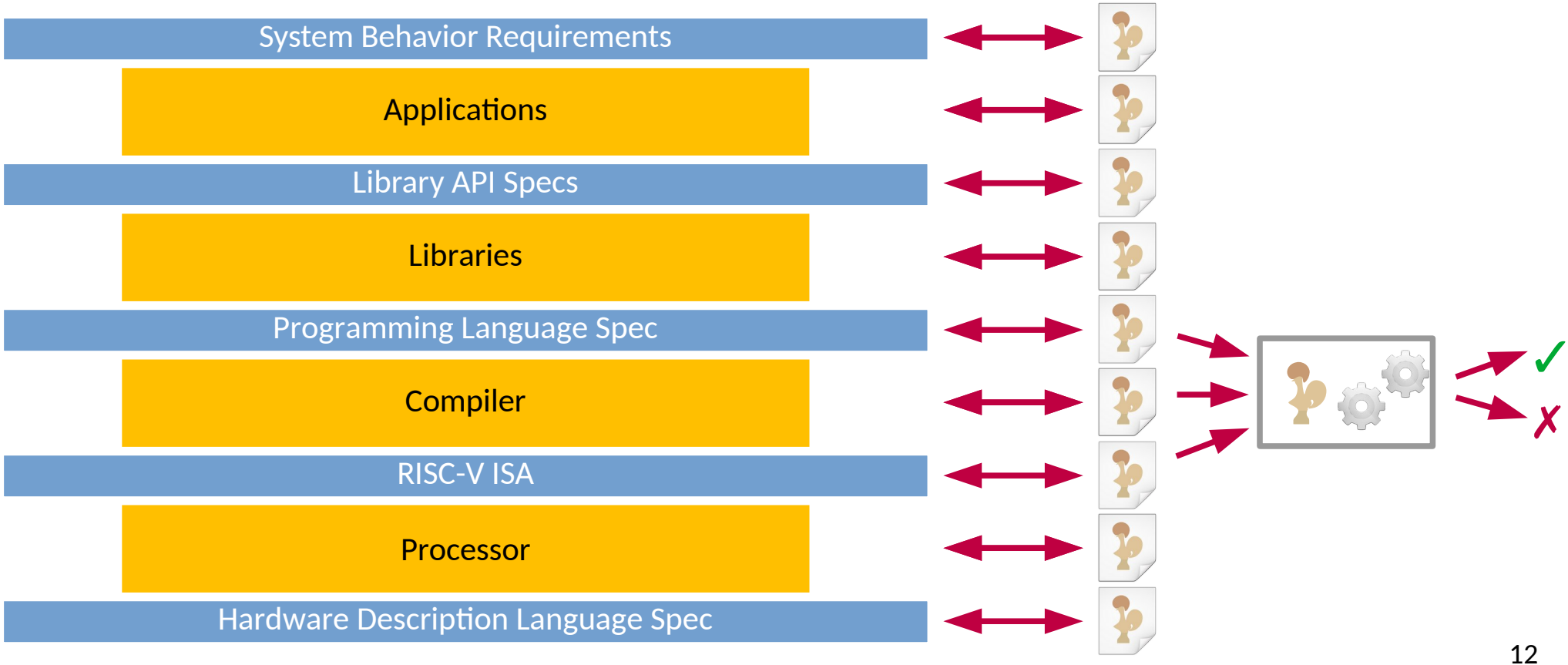
Systems development using machine-checked specs at all levels



Systems development using machine-checked specs at all levels

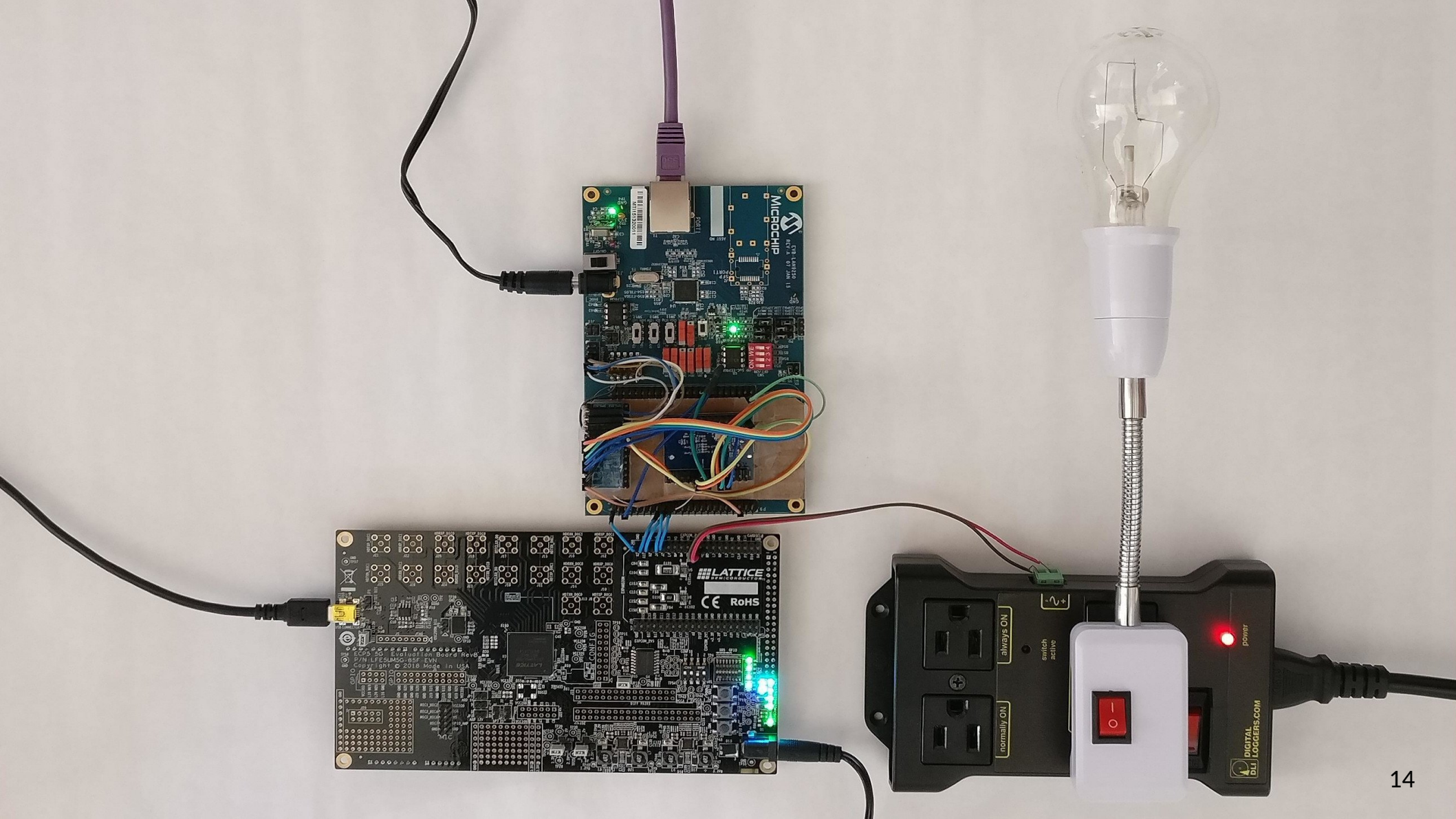


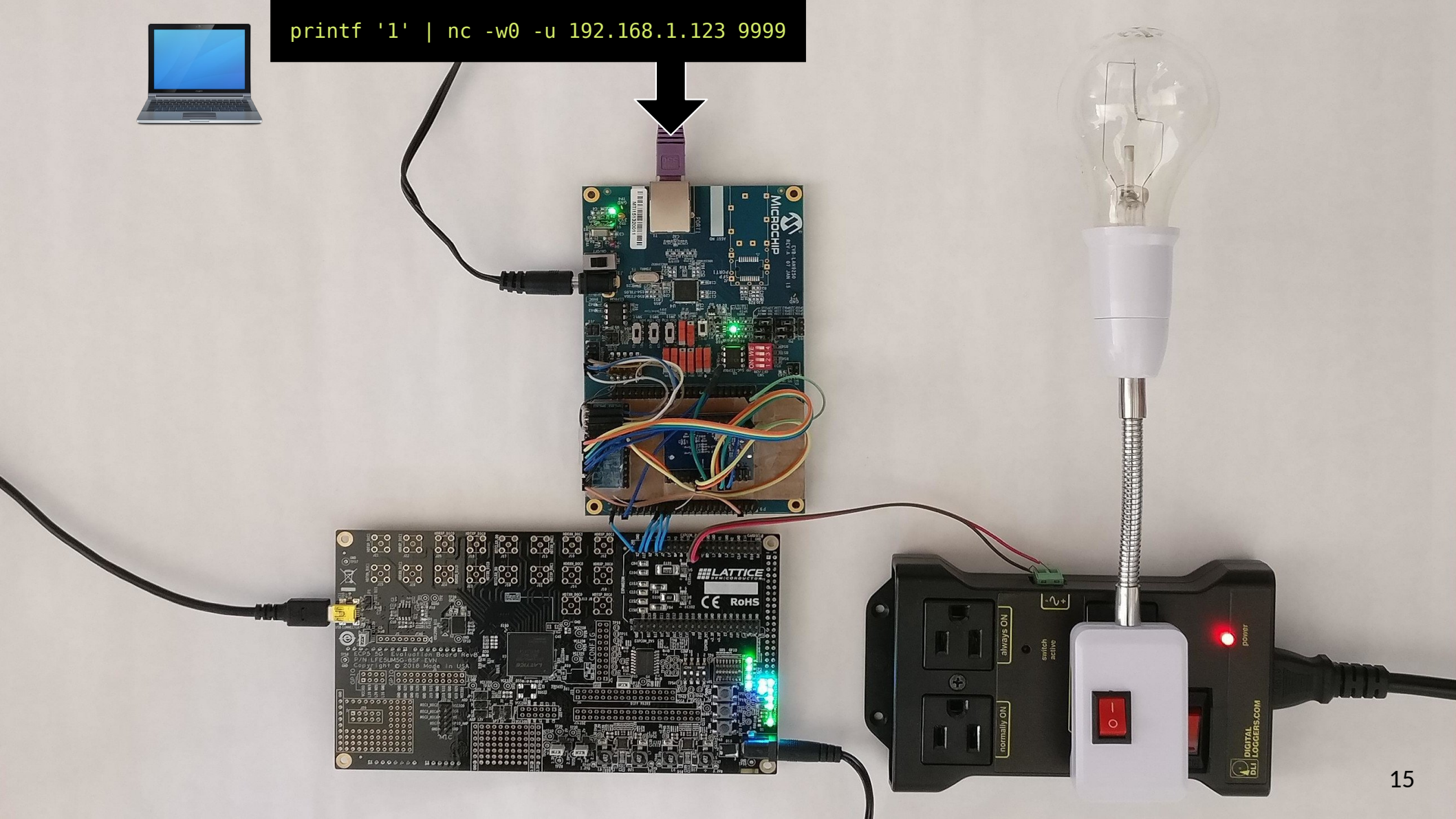
Systems development using machine-checked specs at all levels

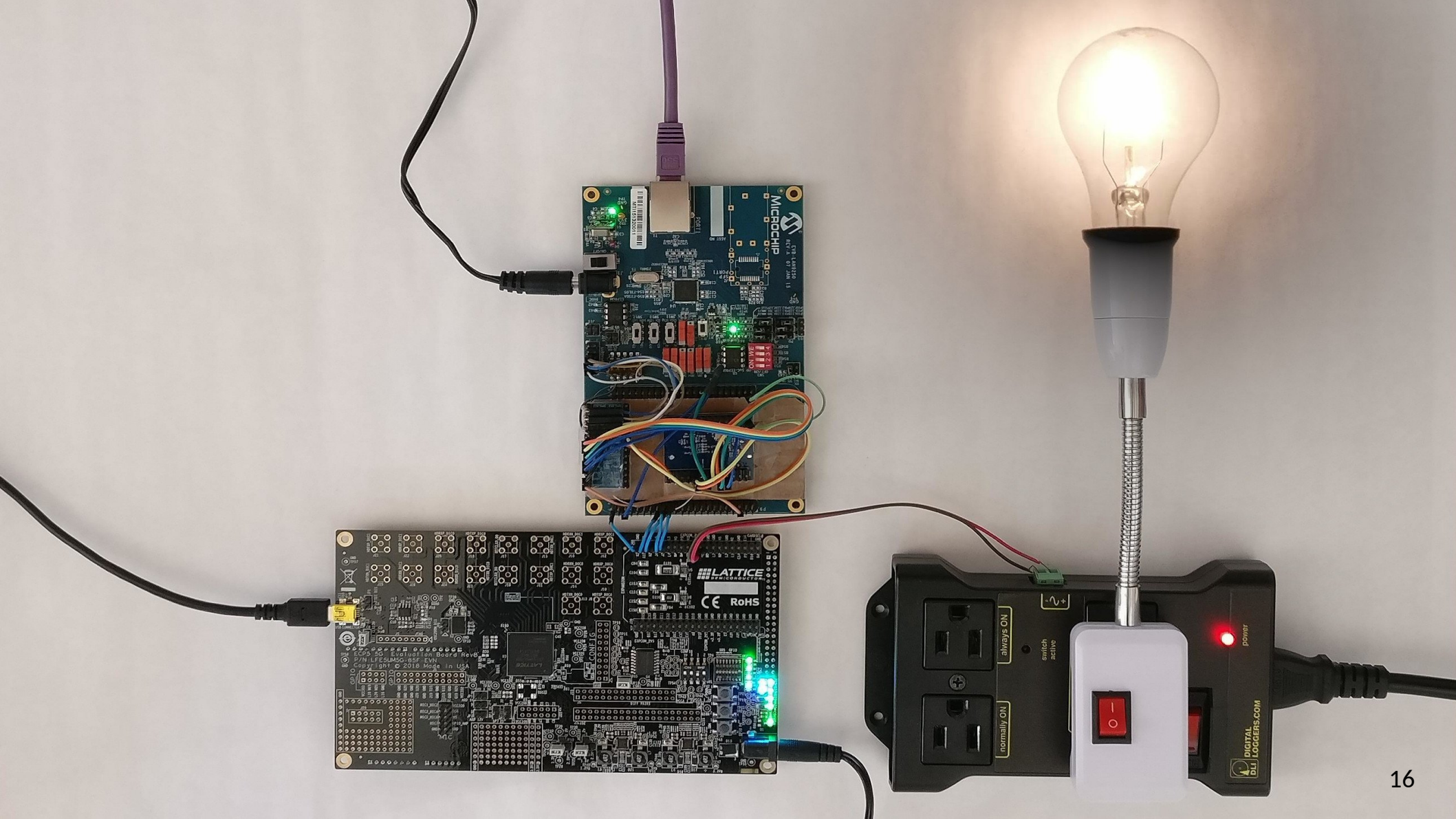


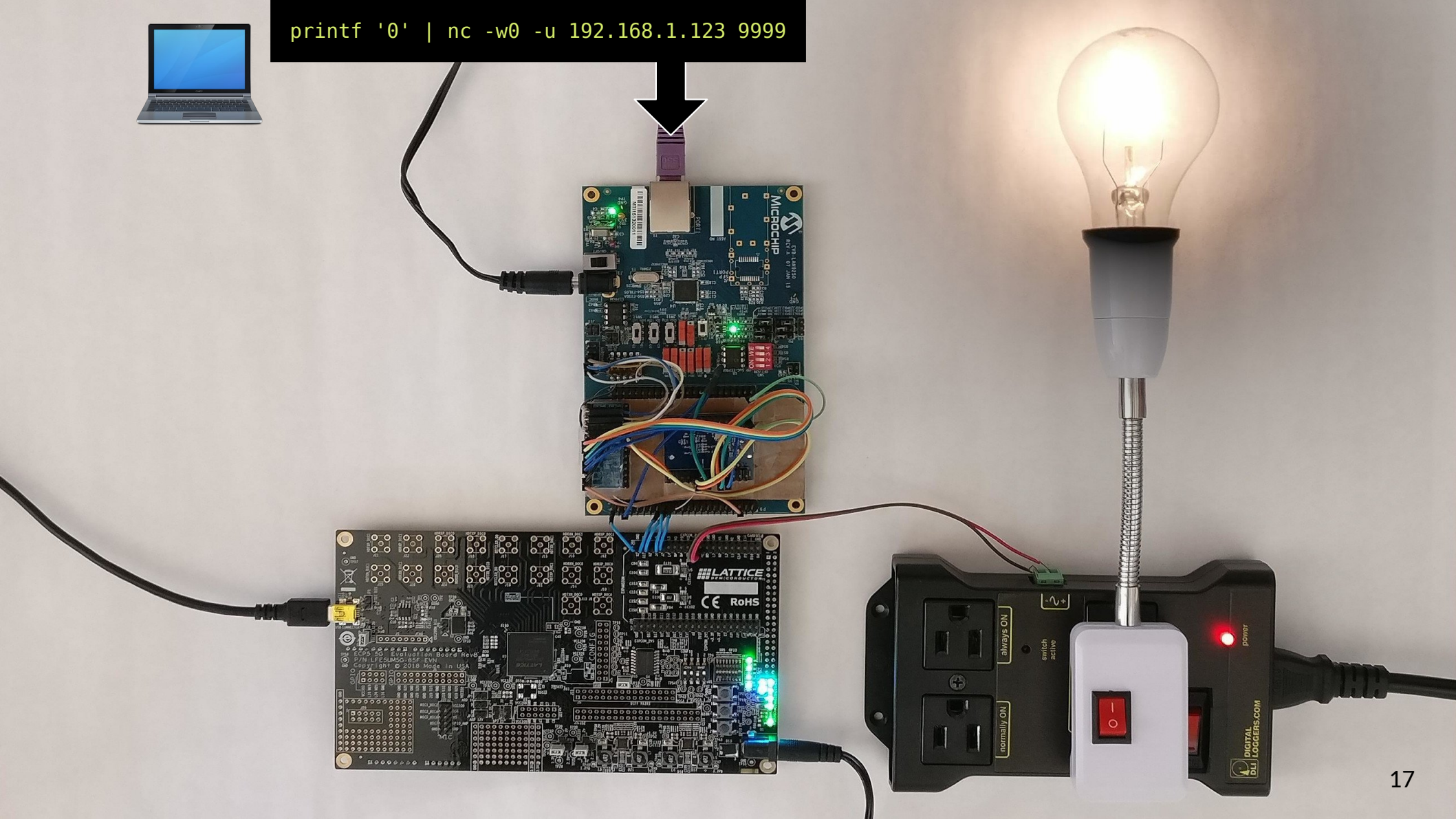
Prototype: The IoT lightbulb

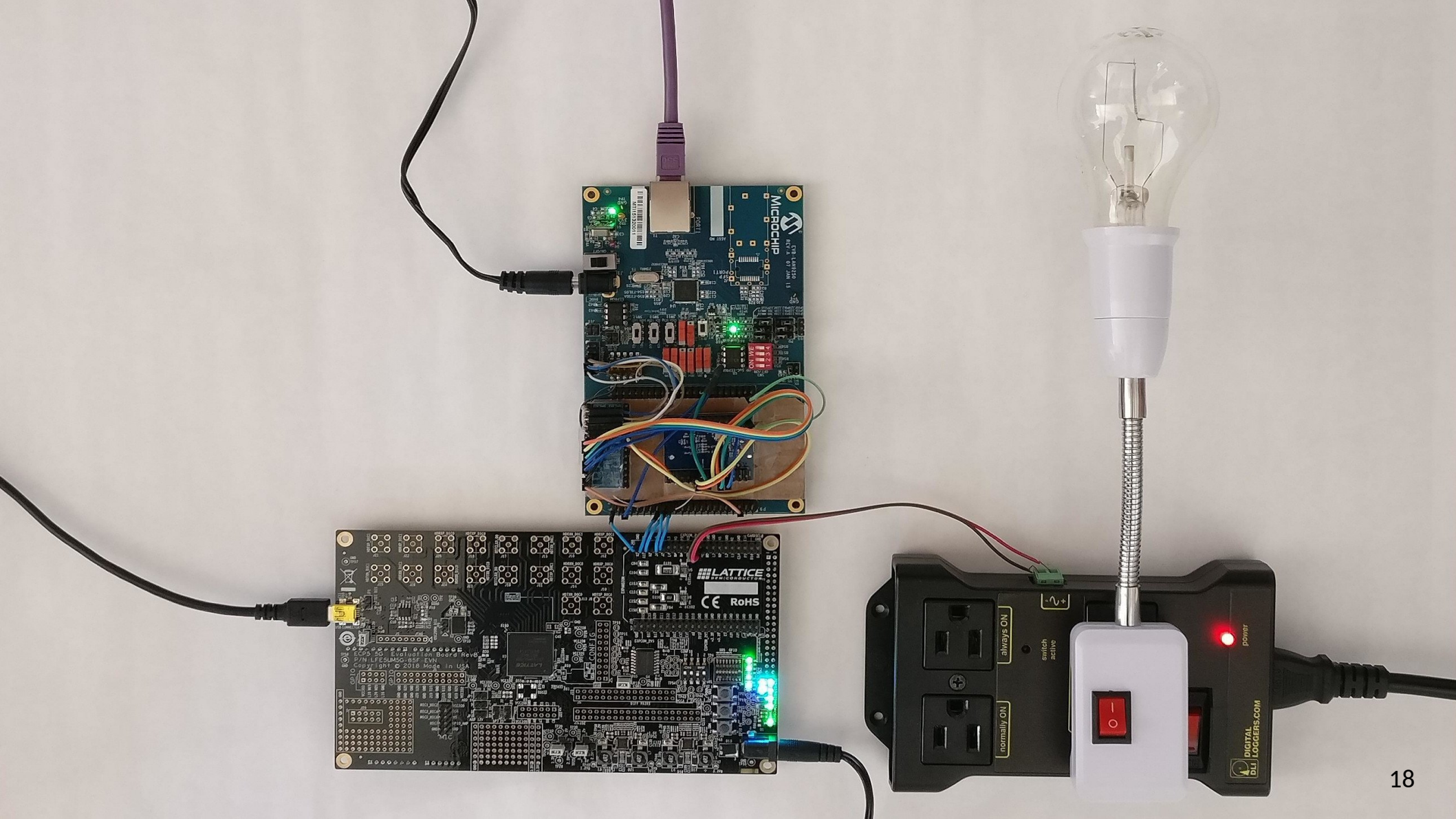
- Wrote all digital parts from scratch
- Using bedrock2
  - a low-level programming language with verification support
- Very simple at the moment
- Goal: Should scale to more realistic applications

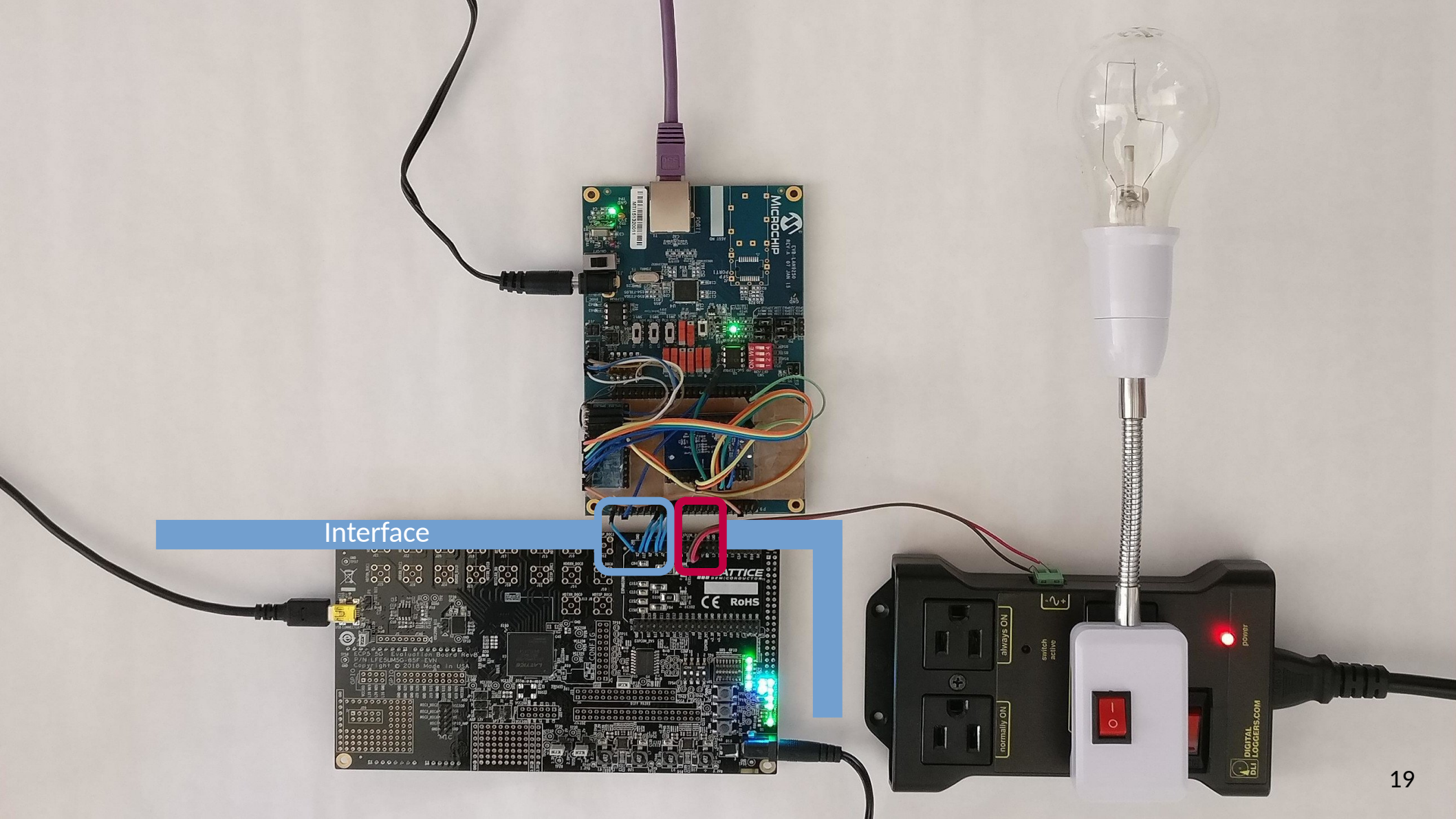

MICROCHIP
LATTICE
RoHS
always ON
normally ON
switch active
power
DIGITAL LOGGERS.COM

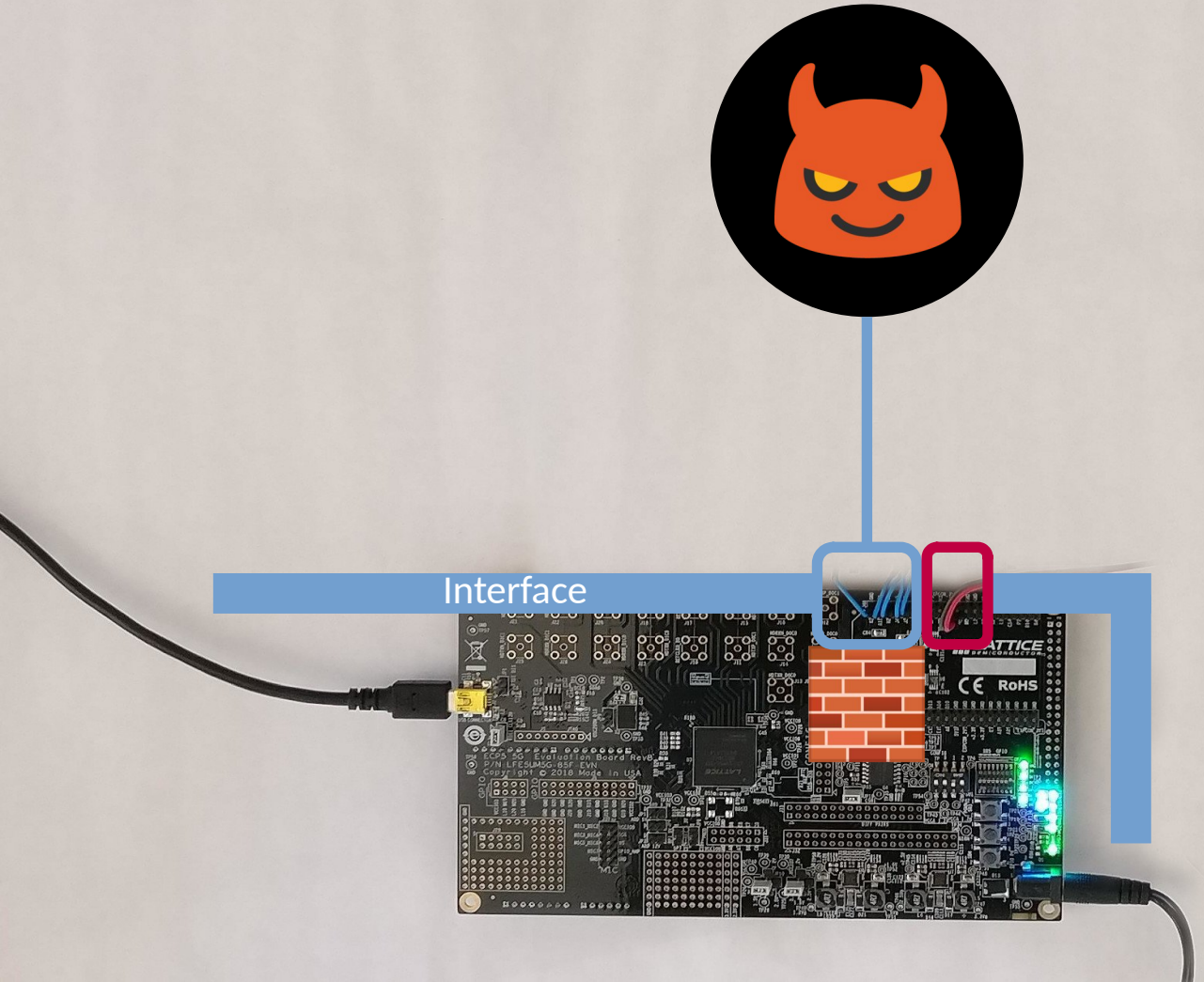

printf '1' | nc -w0 -u 192.168.1.123 9999
MICROCHIP
LATTICE
RoHS
always ON
normally ON
switch active
power
DIGITAL LOGGERS.COM


MICROCHIP
LATTICE
RoHS
always ON
normally ON
switch active
power
DIGITAL LOGGERS.COM

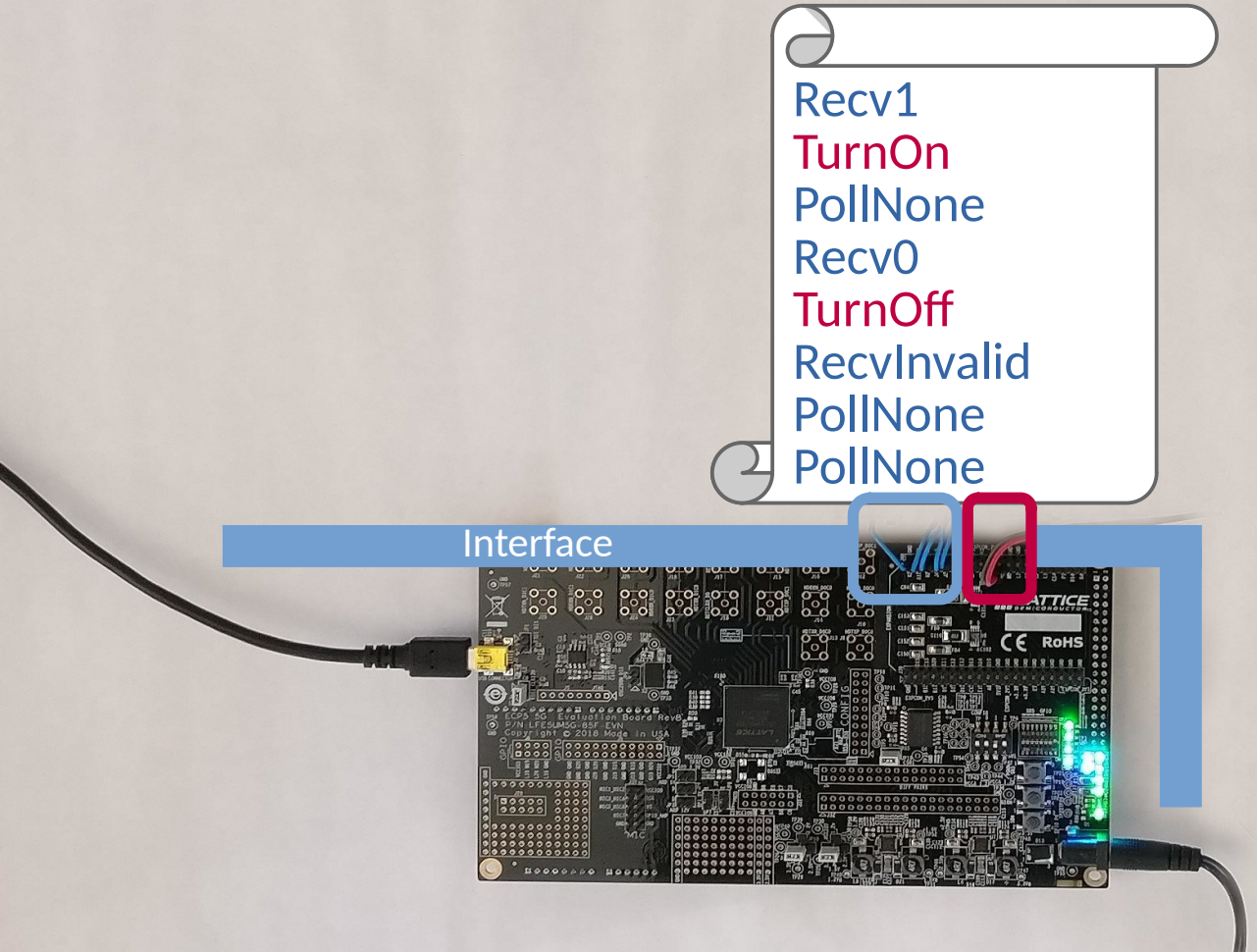

printf '0' | nc -w0 -u 192.168.1.123 9999
MICROCHIP
LATTICE
RoHS
always ON
normally ON
switch active
power
DIGITAL LOGGERS.COM


MICROCHIP
LATTICE
RoHS
always ON
normally ON
switch active
power
DIGITAL LOGGERS.COM


Interface
MICROCHIP
RoHS
always ON
normally ON
switch active
power
DIGITAL LOGGERS.COM


Interface

Specification using a Log



How to specify the behavior of the processor

Small-Step Operational Semantics

step from configuration to configuration:

$$c_0 \longrightarrow c_1 \longrightarrow \ldots \longrightarrow c_n$$

Semantic Rules (1/2)

$$\textsc{LAdd}\ \frac{< e_1, \sigma > \longrightarrow < e_1', \sigma' >}{< e_1 + e_2, \sigma > \longrightarrow < e_1' + e_2, \sigma' >}$$

$$\textsc{RAdd}\ \frac{< e_2, \sigma > \longrightarrow < e_2', \sigma' >}{< n + e_2, \sigma > \longrightarrow < n + e_2', \sigma' >}$$

$$\textsc{Add}\ \frac{}{< n + m, \sigma > \longrightarrow < p, \sigma >}$$ where p is the sum of n and m

$$\textsc{LMul}\ \frac{< e_1, \sigma > \longrightarrow < e_1', \sigma' >}{< e_1 \times e_2, \sigma > \longrightarrow < e_1' \times e_2, \sigma' >}$$

$$\textsc{RMul}\ \frac{< e_2, \sigma > \longrightarrow < e_2', \sigma' >}{< n \times e_2, \sigma > \longrightarrow < n \times e_2', \sigma' >}$$

$$\textsc{Mul}\ \frac{}{< n \times m, \sigma > \longrightarrow < p, \sigma >}$$ where p is the product of n and m

Large-step semantics

$$\Downarrow \subseteq \textbf{Config} \times \textbf{FinalConfig}$$

where

$$\textbf{Config} = \textbf{Exp} \times \textbf{Store}$$
$$\text{and } \textbf{Final Config} = \textbf{Int} \times \textbf{Store} \subseteq \textbf{Config}.$$

We write $< e, \sigma > \Downarrow < n, \sigma' >$ to mean that
$(< e, \sigma >, < n, \sigma' >) \in \Downarrow$

Semantic Rules (1)

$$\textsc{Int}_{\textsc{Lrg}}\ \frac{}{< n, \sigma > \Downarrow < n, \sigma >}$$

$$\textsc{Var}_{\textsc{Lrg}}\ \frac{}{< x, \sigma > \Downarrow < n, \sigma >}\ \text{where } n = \sigma(x)$$

$$\textsc{Add}_{\textsc{Lrg}}\ \frac{\begin{array}{c} < e_1, \sigma > \Downarrow < n_1, \sigma'' > \\ < e_2, \sigma'' > \Downarrow < n_2, \sigma' > \\ \text{where } n \text{ is the sum of } n_1 \text{ and } n_2 \end{array}}{< e_1 + e_2, \sigma > \Downarrow < n, \sigma' >}$$

Arithmetic expressions

$$
\begin{aligned}
\mathcal{A}[\![n]\!] &= \{(\sigma, n)\} \\
\mathcal{A}[\![x]\!] &= \{(\sigma, \sigma(x))\} \\
\mathcal{A}[\![a_1 + a_2]\!] &= \{(\sigma, n) \mid (\sigma, n_1) \in \mathcal{A}[\![a_1]\!] \\
&\qquad \wedge (\sigma, n_2) \in \mathcal{A}[\![a_2]\!] \\
&\qquad \wedge n = n_1 + n_2\} \\
\mathcal{A}[\![a_1 \times a_2]\!] &= \{(\sigma, n) \mid (\sigma, n_1) \in \mathcal{A}[\![a_1]\!] \\
&\qquad \wedge (\sigma, n_2) \in \mathcal{A}[\![a_2]\!] \\
&\qquad \wedge n = n_1 \times n_2\}
\end{aligned}
$$

How to specify the behavior of the processor

- Using the hardware description language Kami
- Embedded in Coq

Kami Semantics

$$
\begin{array}{rrcl}
\text{Expression} & e & ::= & c \mid x \mid \mathrm{op}(\vec{e}) \mid [\vec{e}] \mid \{\overrightarrow{k = e}\} \mid e[e] \mid e.k \\
\text{Action} & a & ::= & \text{let } x = r \text{ in } a \\
 & & \mid & r := e\,;\ a \\
 & & \mid & \text{let } x = f(e) \text{ in } a \\
 & & \mid & \text{let } x = e \text{ in } a \\
 & & \mid & \text{if } e \text{ then } a \text{ else } a\,;\ a \\
 & & \mid & \text{assert } e\,;\ a \\
 & & \mid & \text{return } e \\
\text{Module} & m & ::= & \langle[\overrightarrow{\langle r, c\rangle}], [\overrightarrow{\langle s, a\rangle}], [\overrightarrow{\langle f, \lambda x : \tau.\ a\rangle}]\rangle \\
 & & \mid & m + m
\end{array}
$$

- o: old state of register file (mapping from register names r to values v)
- u: updates to register file
- ℓ: labels (to go in log)

$$\text{ActionReadReg:}\ \frac{o \xrightarrow[[o(r)/x]a]{\ell} (u, v)}{o \xrightarrow[\text{let } x=r \text{ in } a]{\ell} (u, v)}$$

$$\text{ActionWriteReg:}\ \frac{[\![e]\!] = v_r \qquad r \notin u \qquad o \xrightarrow[a]{\ell} (u, v)}{o \xrightarrow[r:=e\,;\ a]{\ell} (u[r \leftarrow v_r], v)}$$

$$\text{ActionCall:}\ \frac{[\![e]\!] = v_a \qquad (f(\_) = \_) \notin \ell \qquad o \xrightarrow[[v_r/x]a]{\ell} (u, v)}{o \xrightarrow[\text{let } x=f(e) \text{ in } a]{\{f(v_a)=v_r\}\cup\ell} (u, v)}$$

Q: Where do you see/expect small-step/big-step/denotational semantics?

More details on Kami: See ICFP'17 paper

Arithmetic expressions

$$
\begin{aligned}
\mathcal{A}[\![n]\!] &= \{(\sigma, n)\} \\
\mathcal{A}[\![x]\!] &= \{(\sigma, \sigma(x))\} \\
\mathcal{A}[\![a_1 + a_2]\!] &= \{(\sigma, n) \mid (\sigma, n_1) \in \mathcal{A}[\![a_1]\!] \\
&\qquad \wedge (\sigma, n_2) \in \mathcal{A}[\![a_2]\!] \\
&\qquad \wedge n = n_1 + n_2\} \\
\mathcal{A}[\![a_1 \times a_2]\!] &= \{(\sigma, n) \mid (\sigma, n_1) \in \mathcal{A}[\![a_1]\!] \\
&\qquad \wedge (\sigma, n_2) \in \mathcal{A}[\![a_2]\!] \\
&\qquad \wedge n = n_1 \times n_2\}
\end{aligned}
$$

Denotational semantics for Kami expressions

$$
\begin{aligned}
[\![c]\!] &= c \\
[\![\mathsf{op}(\vec{e})]\!] &= [\![\mathsf{op}]\!]_{\mathsf{op}}(\overrightarrow{[\![e]\!]}) \\
[\![[e_0, \ldots, e_{2^n-1}]]\!] &= \lambda i : \mathsf{word}(n).\ [\![e_i]\!] \\
[\![\{k_1 = e_{k_1}, \ldots, k_n = e_{k_n}\}]\!] &= \lambda k : \{k_1, \ldots, k_n\}.\ [\![e_k]\!] \\
[\![e_v[e_i]]\!] &= [\![e_v]\!]([\![e_i]\!]) \\
[\![e.k]\!] &= [\![e]\!](k)
\end{aligned}
$$

Inductive Set $\longrightarrow^*$ (Multi-Step Rel.)

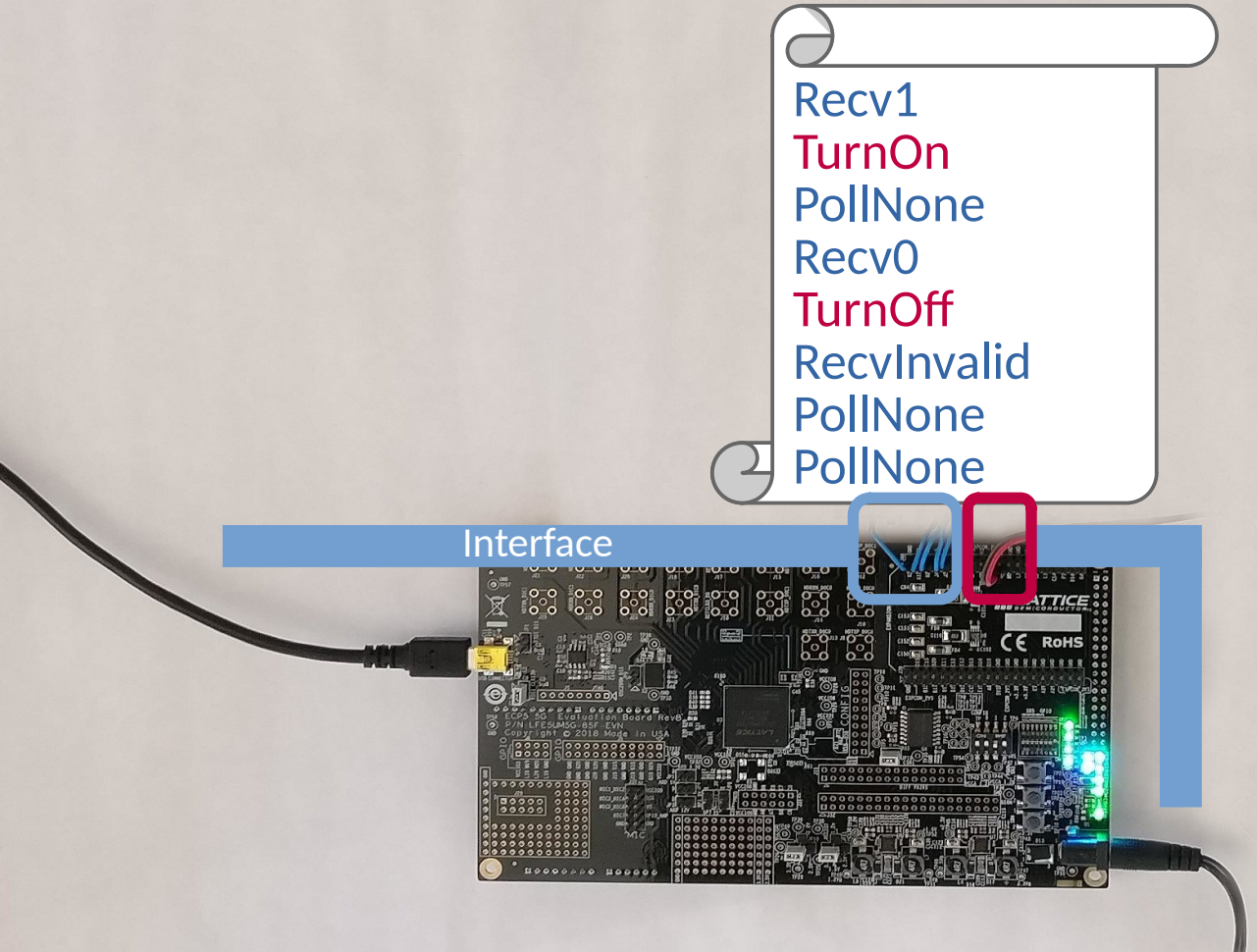
$$\frac{}{< e, \sigma > \longrightarrow^* < e, \sigma >}$$

$$\frac{< e, \sigma > \longrightarrow < e', \sigma' > \qquad < e', \sigma' > \longrightarrow^* < e'', \sigma'' >}{< e, \sigma > \longrightarrow^* < e'', \sigma'' >}$$

From single step to multistep in Kami

```
Inductive Multistep: RegsT -> RegsT -> list LabelT -> Prop :=
| NilMultistep o1 o2: o1 = o2 -> Multistep o1 o2 nil
| Multi o a n (HMultistep: Multistep o n a)
        u l (HStep: Step n u l):
    Multistep o (M.union u n) (l :: a).
```

Specification using a Log



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)*

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:

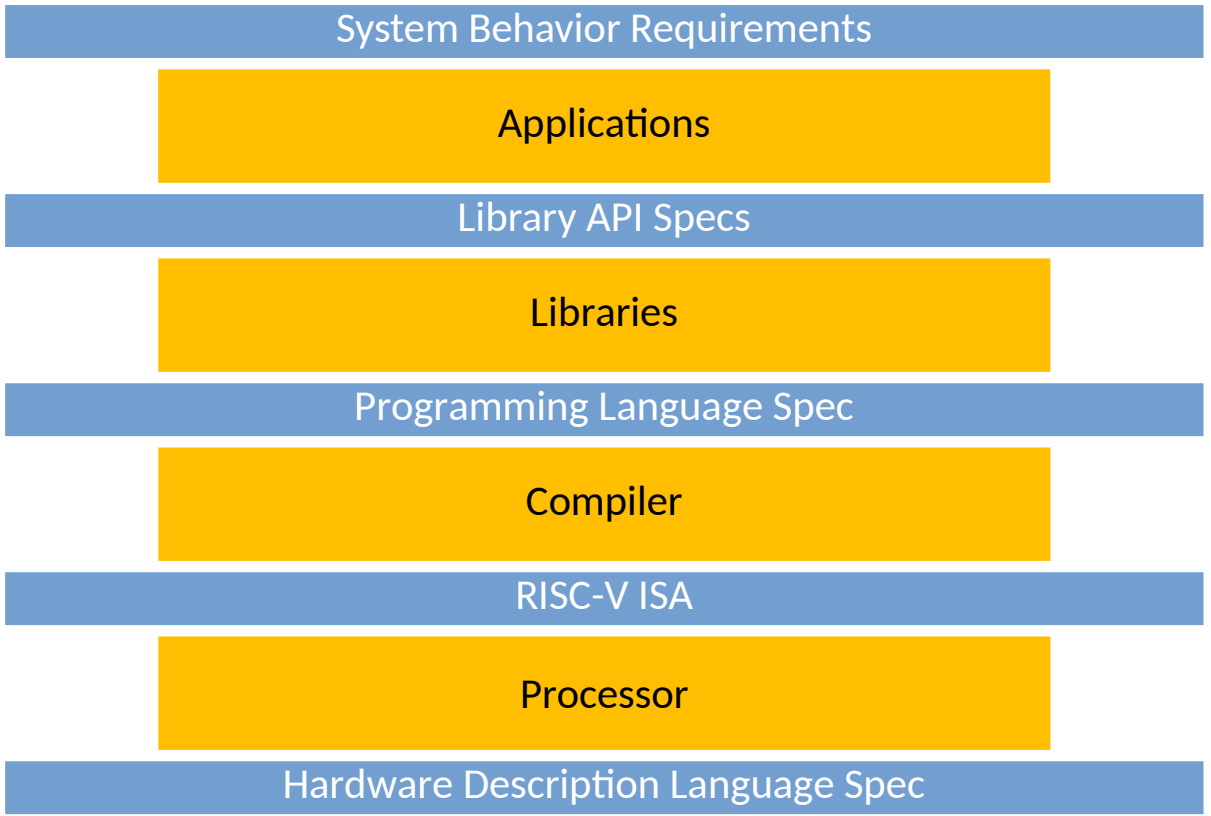
  BootSeq ((Recv1 TurnOn) | (Recv0 TurnOff) | RecvInvalid | PollNone)*

End-to-end theorem

```
Theorem end2end_lightbulb: ∀ mem0 t,
  bytes_at (instrencode lightbulb_insts) 0 mem0 ∧
  Trace (p4mm mem0) t →
  ∃ t': list (string * word * word),
    KamiRiscv.KamiLabelSeqR t t' ∧
    prefix_of t' goodHlTrace.


Definition goodHlTrace := BootSeq +++
  ( (EX b:bool, Recv b +++ LightbulbCmd b)
    ||| RecvInvalid ||| PollNone           )*.
```

The end-to-end theorem



The end-to-end theorem provides

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

The source language

IMP syntax

$$
\begin{aligned}
a ::= \;& x \mid n \mid a_1 + a_2 \mid a_1 \times a_2 \\
b ::= \;& \textbf{true} \mid \textbf{false} \mid a_1 < a_2 \\
c ::= \;& \textbf{skip} \mid x := a \mid c_1 ; c_2 \\
& \mid \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2 \\
& \mid \textbf{while } b \textbf{ do } c
\end{aligned}
$$

Bedrock2 Grammar

e	::=	
	v	integer literal
	x	local variable
	`load`$N(e)$	load N bytes from address e, $N = 1$, 2, or 4
	`loadW`(e)	load 4 or 8 bytes, depending on bitwidth
	e_1 *op* e_2	binary operation, *op* = `+`, `-`, `*`, `&`, ...

c	::=	
	`skip`	do nothing
	$x = e$	assignment to local variable
	`store`$N(e_1, e_2)$	store the lower N bytes of e_2 at address e_1, $N = 1$, 2, or 4
	`storeW`(e_1, e_2)	store 4 or 8 bytes, depending on bitwidth, at address e_1
	`if (`e`)` c_1 `else` c_2	if-then-else
	c_1 `;` c_2	sequence
	`while (`e`)` c	while loop
	$x_1, \ldots x_n = f(e_1, \ldots e_m)$	call procedure f and assign return values to $x_1, \ldots x_n$

Bedrock2 Semantics

`vcgen c t m l P :=`
"what do I need to prove to know that executing command `c` from a state `(t,m,l)` always terminates, and that the final state satisfies `P`?"

```
vcgen skip t m l P :=
   P t m l
vcgen (x = e) t m l P :=
   ∃ v, vcexpr m l e v ∧ P t m l[x:=v]
vcgen (c1; c2) t m l P :=
   vcgen c1 t m l (λ t' m' l' ⇒ vcgen c2 t' m' l' P)
vcgen (store1(e1, e2)) t m l P :=
   ∃ addr, vcexpr m l e1 addr ∧
   ∃ value, vcexpr m l e2 value ∧
   addr ∈ dom(m) ∧
   P t m[addr:=value] l
vcgen (x = f_ext(e)) t m l P :=
   ∃ v, vcexpr m l e v ∧
        vcextern f_ext t [v]
              (λ r ⇒ P ((f,[v],[r])::t) m l[x:=r])
```

Partial Correctness and Total Correctness

$$\{Pre\}\ c\ \{Post\}$$

"If *Pre* holds before *c*, **and** *c* **terminates, then**

Post holds after *c*."

$$[\ Pre\ ]\ c\ [\ Post\ ]$$

"If *Pre* holds before *c* **then** *c* **will terminate** and
Post will hold after *c*."

Sample program verification

- Let’s step through a simple program and see how it creates an MMIO event in the trace

```
function attempt_read_byte() returns (uint32_t err, uint32_t v) {
   uint32_t busy = MMIOREAD(SPI_READ_ADDR);
   if (busy >> 31) {
      err = 1;
   } else {
      err = 0;
      v = busy & 0xff;
   }
}
```

```
Lemma attempt_read_byte_correct: ∀ t m,
   vcgen (body attempt_read_byte) t m empty (λ t' m' l' =>
      ∃b, t' = ("MMIOREAD", SPI_READ_ADDR, b)::t ∧
          (0 <= b < 256 ∧ l'[err] = 0 ∧ l'[v] = b) ∨
          (b >> 31 = 1 ∧ l'[err] = 1).
```

```
∀ t m, vcgen (
   uint32_t busy = MMIOREAD(SPI_READ_ADDR);
   if (busy >> 31) {
      err = 1;
   } else {
      err = 0;
      v = busy & 0xff;
   }
) t m empty (λ t' m' l' =>
      ∃b, t' = ("MMIOREAD", SPI_READ_ADDR, b)::t ∧
          (0 <= b < 256 ∧ l'[err] = 0 ∧ l'[v] = b) ∨
          (b >> 31 = 1 ∧ l'[err] = 1)
```

```
∀ t m, vcgen (
   uint32_t busy = MMIOREAD(SPI_READ_ADDR);
)
t m empty (λ t’ m’ l’ =>
   vcgen (
      if (busy >> 31) {
         err = 1;
      } else {
         err = 0;
         v = busy & 0xff;
      }
   ) t’ m’ l’ (λ t’’ m’’ l’’ =>
      ∃b, t’’ = (“MMIOREAD”, SPI_READ_ADDR, b)::t ∧
          (0 <= b < 256 ∧ l’’[err] = 0 ∧ l’’[v] = b) ∨
          (b >> 31 = 1 ∧ l’’[err] = 1)
)
```

```
∀ t m, vcextern MMIOREAD t [SPI_READ_ADDR] (λ b =>
  (λ t’ m’ l’ =>
      vcgen (
         if (busy >> 31) {
            err = 1;
         } else {
            err = 0;
            v = busy & 0xff;
         }
      ) t’ m’ l’ (λ t’’ m’’ l’’ =>
         ∃b, t’’ = (“MMIOREAD”, SPI_READ_ADDR, b)::t ∧
             (0 <= b < 256 ∧ l’’[err] = 0 ∧ l’’[v] = b) ∨
             (b >> 31 = 1 ∧ l’’[err] = 1).
   ) ((“MMIOREAD”, SPI_READ_ADDR, b)::t) m empty[busy:=b])
```

```
∀ t m, vcextern MMIOREAD t [SPI_READ_ADDR] (λ b =>
   vcgen
   (
      if (busy >> 31) {
         err = 1;
      } else {
         err = 0;
         v = busy & 0xff;
      }
   )
   ((“MMIOREAD”, SPI_READ_ADDR, b)::t)
   m
   empty[busy:=b]
   (λ t’’ m’’ l’’ =>
         ∃b, t’’ = (“MMIOREAD”, SPI_READ_ADDR, b)::t ∧
             (0 <= b < 256 ∧ l’’[err] = 0 ∧ l’’[v] = b) ∨
             (b >> 31 = 1 ∧ l’’[err] = 1)))
```

```
∀ t m, SPI_READ_ADDR ∈ MMIO_RANGE ∧ ∀ r, (λ b =>
   vcgen
   (
      if (busy >> 31) {
         err = 1;
      } else {
         err = 0;
         v = busy & 0xff;
      }
   )
   (("MMIOREAD", SPI_READ_ADDR, b)::t)
   m
   empty[busy:=b]
   (λ t'' m'' l'' =>
         ∃b, t'' = ("MMIOREAD", SPI_READ_ADDR, b)::t ∧
             (0 <= b < 256 ∧ l''[err] = 0 ∧ l''[v] = b) ∨
             (b >> 31 = 1 ∧ l''[err] = 1))
) r
```

```
∀ t m, SPI_READ_ADDR ∈ MMIO_RANGE ∧ ∀ r,
   vcgen
   (
      if (busy >> 31) {
         err = 1;
      } else {
         err = 0;
         v = busy & 0xff;
      }
   )
   ((“MMIOREAD”, SPI_READ_ADDR, r)::t)
   m
   empty[busy:=r]
   (λ t’’ m’’ l’’ =>
         ∃b, t’’ = (“MMIOREAD”, SPI_READ_ADDR, b)::t ∧
             (0 <= b < 256 ∧ l’’[err] = 0 ∧ l’’[v] = b) ∨
             (b >> 31 = 1 ∧ l’’[err] = 1))
```

```
∀ t m r,
   vcgen
   (
      if (busy >> 31) {
         err = 1;
      } else {
         err = 0;
         v = busy & 0xff;
      }
   )
   (("MMIOREAD", SPI_READ_ADDR, r)::t)
   m
   empty[busy:=r]
   (λ t'' m'' l'' =>
         ∃b, t'' = ("MMIOREAD", SPI_READ_ADDR, b)::t ∧
             (0 <= b < 256 ∧ l''[err] = 0 ∧ l''[v] = b) ∨
             (b >> 31 = 1 ∧ l''[err] = 1))
```

```
∀ t m r,
   ∃ a, vcexpr m empty[busy:=r] (busy >> 31) a
      ∧ (a <> 0 ->
         vcgen (
            err = 1
         ) ((“MMIOREAD”, SPI_READ_ADDR, r)::t) m empty[busy:=r]
         (λ t’’ m’’ l’’ =>
               ∃b, t’’ = (“MMIOREAD”, SPI_READ_ADDR, b)::t ∧
                   (0 <= b < 256 ∧ l’’[err] = 0 ∧ l’’[v] = b) ∨
                   (b >> 31 = 1 ∧ l’’[err] = 1)))
      ∧ (a = 0 ->
         vcgen (
            err = 0;
            v = busy & 0xff;
         ) ((“MMIOREAD”, SPI_READ_ADDR, r)::t) m empty[busy:=r]
         (λ t’’ m’’ l’’ =>
               ∃b, t’’ = (“MMIOREAD”, SPI_READ_ADDR, b)::t ∧
                   (0 <= b < 256 ∧ l’’[err] = 0 ∧ l’’[v] = b) ∨
                   (b >> 31 = 1 ∧ l’’[err] = 1)))
```

```
∀ t m r,
   ∃ a, a = r >> 31
      ∧ (a <> 0 ->
         vcgen (
            err = 1
         ) ((“MMIOREAD”, SPI_READ_ADDR, r)::t) m empty[busy:=r]
         (λ t’’ m’’ l’’ =>
               ∃b, t’’ = (“MMIOREAD”, SPI_READ_ADDR, b)::t ∧
                   (0 <= b < 256 ∧ l’’[err] = 0 ∧ l’’[v] = b) ∨
                   (b >> 31 = 1 ∧ l’’[err] = 1)))
      ∧ (a = 0 ->
         vcgen (
            err = 0;
            v = busy & 0xff;
         ) ((“MMIOREAD”, SPI_READ_ADDR, r)::t) m empty[busy:=r]
         (λ t’’ m’’ l’’ =>
               ∃b, t’’ = (“MMIOREAD”, SPI_READ_ADDR, b)::t ∧
                   (0 <= b < 256 ∧ l’’[err] = 0 ∧ l’’[v] = b) ∨
                   (b >> 31 = 1 ∧ l’’[err] = 1)))
```

```
∀ t m r,
        (r >> 31 <> 0 ->
         vcgen (
            err = 1
         ) ((“MMIOREAD”, SPI_READ_ADDR, r)::t) m empty[busy:=r]
         (λ t’’ m’’ l’’ =>
               ∃b, t’’ = (“MMIOREAD”, SPI_READ_ADDR, b)::t ∧
                   (0 <= b < 256 ∧ l’’[err] = 0 ∧ l’’[v] = b) ∨
                   (b >> 31 = 1 ∧ l’’[err] = 1)))
      ∧ (r >> 31 = 0 ->
         vcgen (
            err = 0;
            v = busy & 0xff;
         ) ((“MMIOREAD”, SPI_READ_ADDR, r)::t) m empty[busy:=r]
         (λ t’’ m’’ l’’ =>
               ∃b, t’’ = (“MMIOREAD”, SPI_READ_ADDR, b)::t ∧
                   (0 <= b < 256 ∧ l’’[err] = 0 ∧ l’’[v] = b) ∨
                   (b >> 31 = 1 ∧ l’’[err] = 1)))
```

```
∀ t m r,
        (r >> 31 <> 0 ->
         vcgen (
            err = 1
         ) ((“MMIOREAD”, SPI_READ_ADDR, r)::t) m empty[busy:=r]
         (λ t’’ m’’ l’’ =>
               ∃b, t’’ = (“MMIOREAD”, SPI_READ_ADDR, b)::t ∧
                   (0 <= b < 256 ∧ l’’[err] = 0 ∧ l’’[v] = b) ∨
                   (b >> 31 = 1 ∧ l’’[err] = 1)))
```

```
∀ t m r,
        (r >> 31 <> 0 ->
         (λ t'' m'' l'' =>
               ∃b, t'' = (“MMIOREAD”, SPI_READ_ADDR, b)::t ∧
                   (0 <= b < 256 ∧ l''[err] = 0 ∧ l''[v] = b) ∨
                   (b >> 31 = 1 ∧ l''[err] = 1)))
         ((“MMIOREAD”, SPI_READ_ADDR, r)::t) m empty[busy:=r][err:=1]
```

```
∀ t m r, r >> 31 <> 0 ->
  ∃b, ((“MMIOREAD”, SPI_READ_ADDR, r)::t) = (“MMIOREAD”, SPI_READ_ADDR, b)::t ∧
      ((0 <= b < 256 ∧ empty[busy:=r][err:=1][err] = 0 ∧
       empty[busy:=r][err:=1][v] = b)
       ∨
      (b >> 31 = 1 ∧ empty[busy:=r][err:=1][err] = 1)))
```

```
∀ t m r, r >> 31 <> 0 ->
      True ∧
      ((0 <= r < 256 ∧ empty[busy:=r][err:=1][err] = 0 ∧
       empty[busy:=r][err:=1][v] = r)
       ∨
      (r >> 31 = 1 ∧ empty[busy:=r][err:=1][err] = 1)))
```

```
∀ t m r, r >> 31 <> 0 ->
      (r >> 31 = 1 ∧ empty[busy:=r][err:=1][err] = 1)
```

```
∀ t m r, r >> 31 <> 0 ->
       r >> 31 = 1 ∧ empty[busy:=r][err:=1][err] = 1
```

Viewing postconditions as continuations

$$\textsc{Seq}\ \frac{\vdash \{P\}\ c_1\ \{R\} \qquad \vdash \{R\}\ c_2\ \{Q\}}{\vdash \{P\}\ c_1; c_2\ \{Q\}}$$

$$\mathcal{T}[\![\mathit{let}\, x = e_1\ \mathit{in}\, e_2]\!] = \lambda k.\, \mathcal{T}[\![e_1]\!]\ (\lambda x.\, \mathcal{T}[\![e_2]\!] k)$$

k: continuation taking the result value of the expression
P: postcondition taking resulting trace t, memory m, and local vars l

```
vcgen (c1; c2) t m l P :=
   vcgen c1 t m l (λ t' m' l' ⇒ vcgen c2 t' m' l' P)
```

Bedrock2 semantics for compiler correctness proof

- vcgen is a function
- But for compiler correctness, need an Inductive representing executions to drive proof by induction over executions of programs

EVAL-UNOP

$$\frac{(y, v_y) \in \ell \qquad \text{evalunop}(op, v_y, v) \qquad Q(m, \ell[x := v], \tau)}{(x = op(y))/m/\ell/\tau \Downarrow Q}$$

EVAL-INPUT

$$\frac{\forall n, Q(m, \ell[x := n], \tau :: \mathsf{IN}\ n)}{(x = \text{input}())/m/\ell/\tau \Downarrow Q}$$

EVAL-SEQ

$$\frac{c_1/m/\ell/\tau \Downarrow Q_1 \qquad (\forall m'\ \ell'\ \tau',\ Q_1(m', \ell', \tau') \implies c_2/m'/\ell'/\tau' \Downarrow Q)}{(c_1; c_2)/m/\ell/\tau \Downarrow Q}$$

EVAL-STORE

$$\frac{(x, a) \in \ell \qquad (a + n) \in \text{dom}\ m \qquad (y, v) \in \ell \qquad Q(m[(a + n) := v], \ell, \tau)}{(x[n] = y)/m/\ell/\tau \Downarrow Q}$$

Large-step semantics

$$\Downarrow \subseteq \textbf{Config} \times \textbf{FinalConfig}$$

where

$$\textbf{Config} = \textbf{Exp} \times \textbf{Store}$$
$$\text{and } \textbf{Final Config} = \textbf{Int} \times \textbf{Store} \subseteq \textbf{Config}.$$

We write $< e, \sigma > \Downarrow < n, \sigma' >$ to mean that
$(< e, \sigma >, < n, \sigma' >) \in \Downarrow$

Compiler correctness

Simple case: Source and target language is the same

$\forall$ t m l, c/m/l/t $\Downarrow$ Q $\longrightarrow$ compile(c)/m/l/t $\Downarrow$Q

"c/m/l/t $\Downarrow$ Q" is an Inductive Prop, and we can write proofs by induction on it

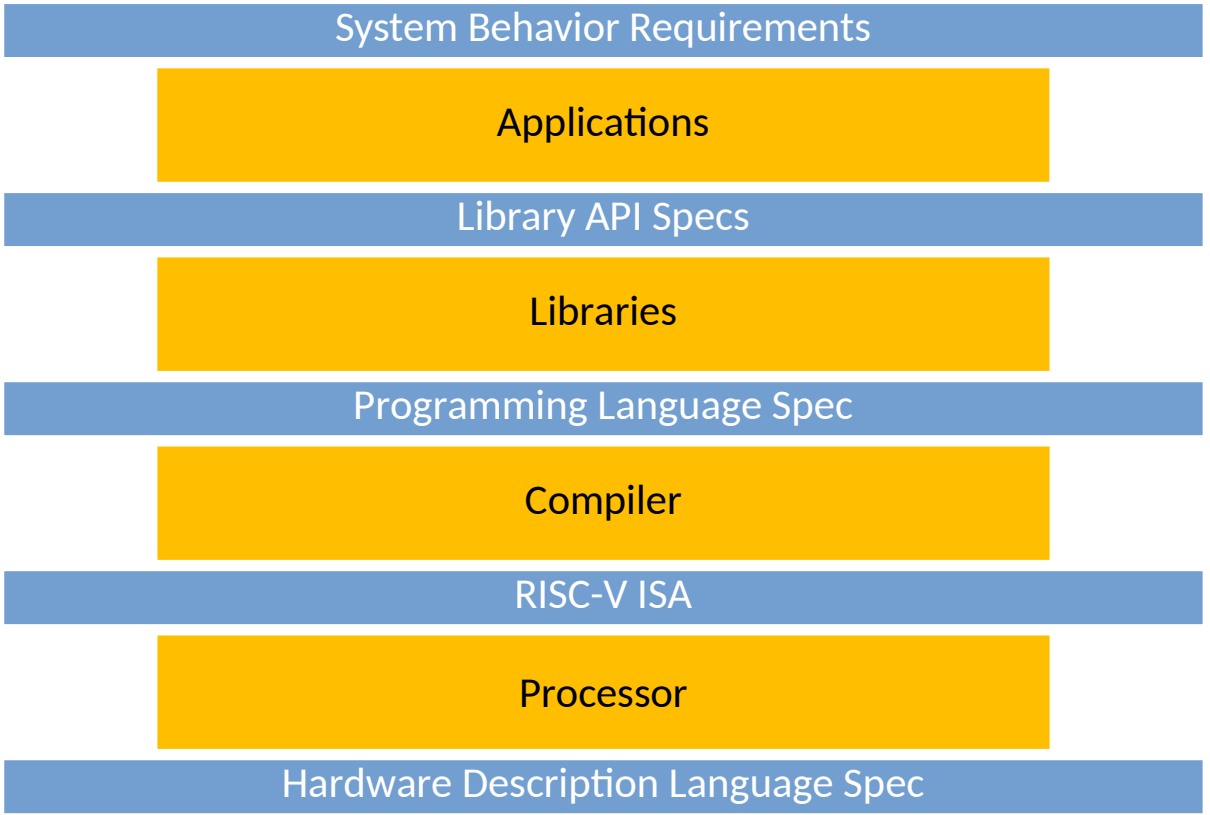
Omnisemantics: Smoother Handling of Nondeterminism

ARTHUR CHARGUÉRAUD, Inria & Université de Strasbourg, CNRS, ICube, France
ADAM CHLIPALA, MIT CSAIL, USA
ANDRES ERBSEN, MIT CSAIL, USA
SAMUEL GRUETTER, MIT CSAIL, USA

This paper gives an in-depth presentation of the omni-big-step and omni-small-step styles of semantic judgments. These styles describe operational semantics by relating starting states to sets of outcomes rather than to individual outcomes. A single derivation of these semantics for a particular starting state and program describes all possible nondeterministic executions (hence the name *omni*), whereas in traditional small-step and big-step semantics, each derivation only talks about one single execution. This restructuring allows for straightforward modeling of languages featuring both nondeterminism and undefined behavior. Specifically, omnisemantics inherently assert *safety*, i.e. they guarantee that none of the execution branches gets stuck, while traditional semantics need either a separate judgment or additional error markers to specify safety in the presence of nondeterminism.

Omnisemantics can be understood as an inductively defined weakest-precondition semantics (or more generally, predicate-transformer semantics) that does not involve invariants for loops and recursion, but instead uses unrolling rules like in traditional small-step and big-step semantics. Omnisemantics have already been used in the past, but we believe that it has been under-appreciated and that it deserves a well-motivated, extensive and pedagogical presentation of its benefits. We also explore several novel aspects associated with these semantics, in particular their use in type-soundness proofs for lambda calculi, partial-correctness reasoning, and forward proofs of compiler correctness for terminating but potentially nondeterministic programs being compiled to nondeterministic target languages. All results in this paper are formalized in Coq.

Apply semantics, PL, verification throughout the whole stack



Questions?

References:

- Integration Verification across Software and Hardware for a Simple Embedded System
  https://dl.acm.org/doi/pdf/10.1145/3453483.3454065
- Omnisemantics: Smoother Handling of Nondeterminism
  https://hal.inria.fr/hal-03255472v2/document
- A Multipurpose Formal RISC-V Specification
  https://arxiv.org/pdf/2104.00762.pdf
- Kami: a platform for high-level parametric hardware specification and its modular verification
  https://dl.acm.org/doi/pdf/10.1145/3110268
- Coq Code
  https://github.com/mit-plv/bedrock2

✉ gruetter@mit.edu

@samuelgruetter