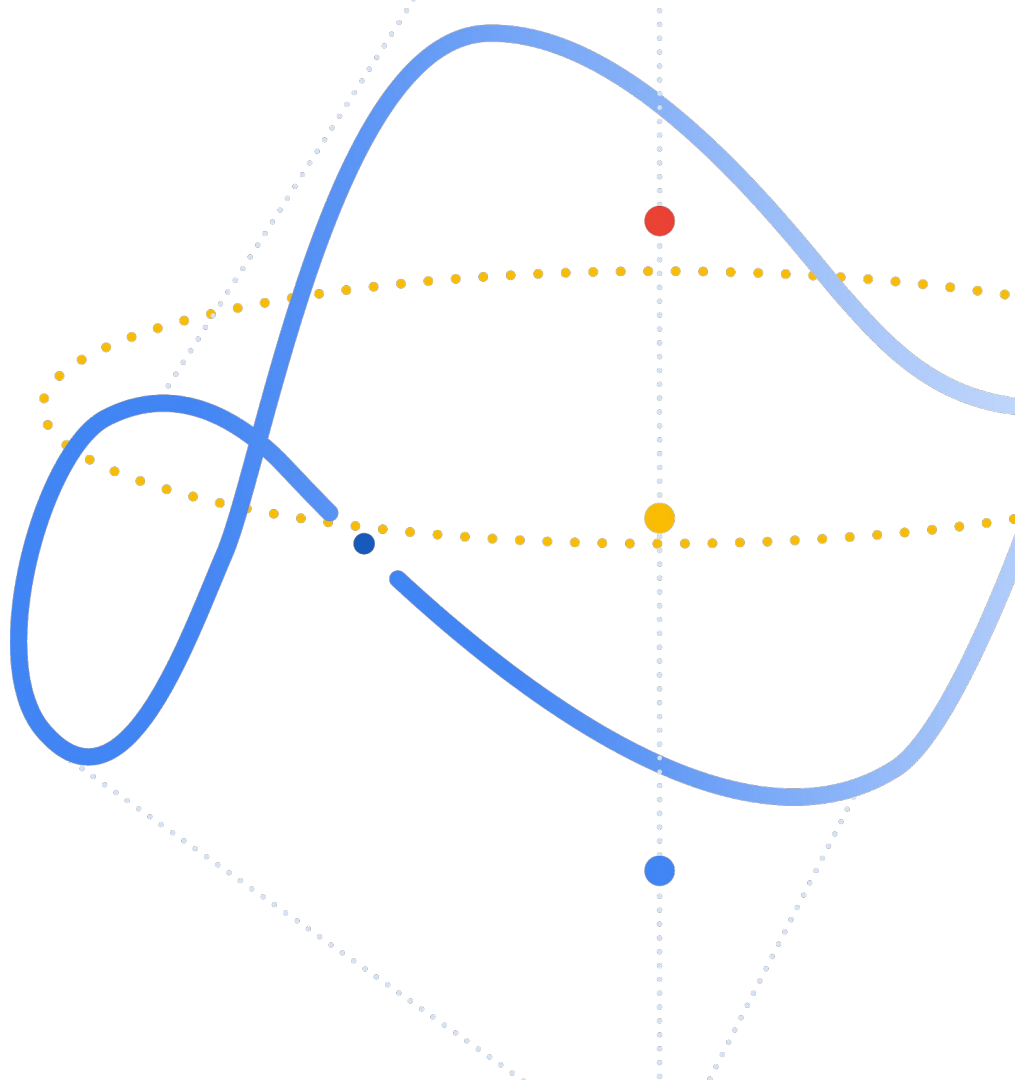


# Hardware Design and Verification with Cava

Satnam Singh  
[satnam@google.com](mailto:satnam@google.com)



## Silver Oak Team



**Ben Blaxill**



**Jade Philipoom**



**Dayeol Lee  
(Berkley)**



**Samuel Gruetter  
(MIT)**

README.md



## About

Meaningful control of data in distributed systems.

[distributed-systems](#)[enclave](#)[policy](#)[formal-methods](#)

# Project Oak

[BUILD](#)[STATUS](#)[DOCS](#)[RUST](#)[SLACK](#)[PROJECT-OAK](#)[MAILING LIST](#)[PROJECT-OAK-DISCUSS](#)

The goal of Project Oak is to create a specification and a reference implementation for the secure transfer, storage and processing of data.

In present computing platforms (including virtualized, and cloud platforms), data may be encrypted at rest and in transit, but they are exposed to any part of the system that needs to process them. Even if the application is securely designed and data are encrypted, the operating system kernel (and any component with privileged access to the machine that handles the data) has unrestricted access to the machine hardware resources, and can leverage that to bypass any security mechanism on the machine itself and extract secret keys and data.

As part of Project Oak, data are end-to-end encrypted between *enclaves*, which are isolated computation compartments that can be created on-demand, and provide strong confidentiality, integrity, and attestation capabilities via a combination of hardware and software functionality. Enclaves protect data and code even from the operating system kernel and privileged software, and are intended to protect from most hardware attacks.

Additionally, data are associated with policies when they enter the system, and policies are enforced and propagated as data move from enclave to enclave.

main ▾

20 branches

0 tags

Go to file

Add file ▾

Code ▾



blaxill Cache third\_party based on submodule commits (#820) ...

✓ 66503b7 2 days ago ⌚ 879 commits

.github/workflows	Cache third_party based on submodule commits (#820)	2 days ago
cava	Circuit equivalence relation (#813)	7 days ago
demos	Refactor circuit simulation definition (#805)	10 days ago
docs	Refactor circuit simulation definition (#805)	10 days ago
examples	Fix a tiny typo (#796)	21 days ago

## About

Formal specification and verification of hardware, especially for security and privacy.

[hardware](#) [coq](#) [formal-verification](#)

Readme

Apache-2.0 License

README.md

## Silver Oak

Silver Oak is a research project at Google Research exploring alternative techniques for producing high assurance circuits and systems based on an approach that unifies specification, implementation and formal verification in a single system, specifically the [Coq](#) interactive theorem prover. We follow an approach inspired by the vision set out by [Adam Chlipala](#) at MIT in his book [Certified Programming with Dependent Types](#).

The Silver Oak project focuses on the design and verification of high assurance variants of some of the peripherals used in the [OpenTitan](#) silicon root of trust e.g. the AES crypto-accelerator block. We focus on the specification, implementation and verification of low-level structural circuits built bottom up by composing basic circuit elements (gates, registers, wires) using powerful higher order combinators in the style of [Lava](#). Another Coq-based approach for producing hardware is [Kami](#) which encodes aspects of the [Bluespec](#) hardware description language as a EDSL in Coq. Kami and Bluespec are powerful tools for designing processor-style control-orientated circuits. We focus instead on "network-style" and "datapath" low level circuits e.g. hardware accelerators for AES.

# Interactive Online Cava Tutorial

<https://project-oak.github.io/silveroak/demo/tutorial.html>

Built with [Alectryon](#), running Coq+SerAPI v8.13.0+0.13.0. Bubbles (◀) indicate interactive fragments: hover for details, tap to reveal contents. Use `Ctrl+*` `Ctrl+↓` to navigate, `Ctrl+*` to focus. On Mac, use `⌘` instead of `Ctrl`. Style: centered; floating; windowed.

## Tutorial

Welcome! This is a quick primer for designing circuits with the Cava DSL. This tutorial will not explain Coq syntax in depth, but will use the same few patterns throughout; you shouldn't need to be a Coq expert to follow along. We'll walk through a few small examples end-to-end, showing you how to define, simulate, and generate netlists for circuits in Cava.

This page (thanks to the [Alectryon](#) system) allows you to see the Coq output for each line that has output. Try hovering over the following line (if on mobile, tap the line):

**Compute** (1 + 2). =

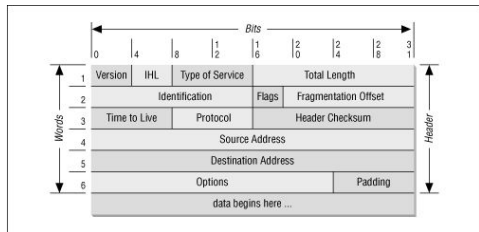
```
= 3
: nat
```

See the banner at the top of the page for instructions on how to navigate the proofs.

### Table of Contents

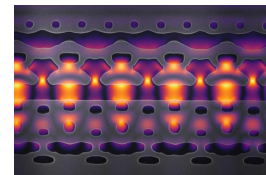
- [Setup](#)
- [Example 1 : Inverter](#)
- [Example 2 : Byte xor](#)
- [Example 3 : Bit-vector xor](#)
- [Example 4 : Tree of xors](#)
- [Example 5 : Delay for Three Timesteps](#)
- [Example 6 : Sum the Input Stream](#)
- [Example 7 : Fibonacci Sequence](#)

# The Problem



data encrypted in transit

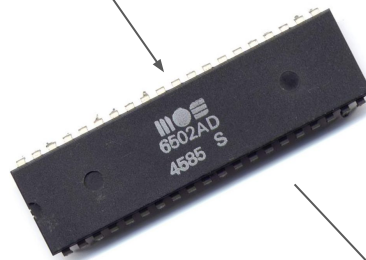
data encrypted at rest



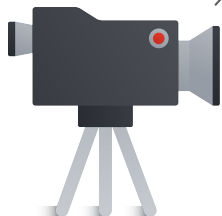
attached accelerators (GPU, crypto, ML)

???

unrestricted access



data encrypted at rest

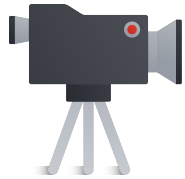


privileged access



# Computing in a secure enclave (Compute average without disclosing individual numbers)

data encrypted at rest



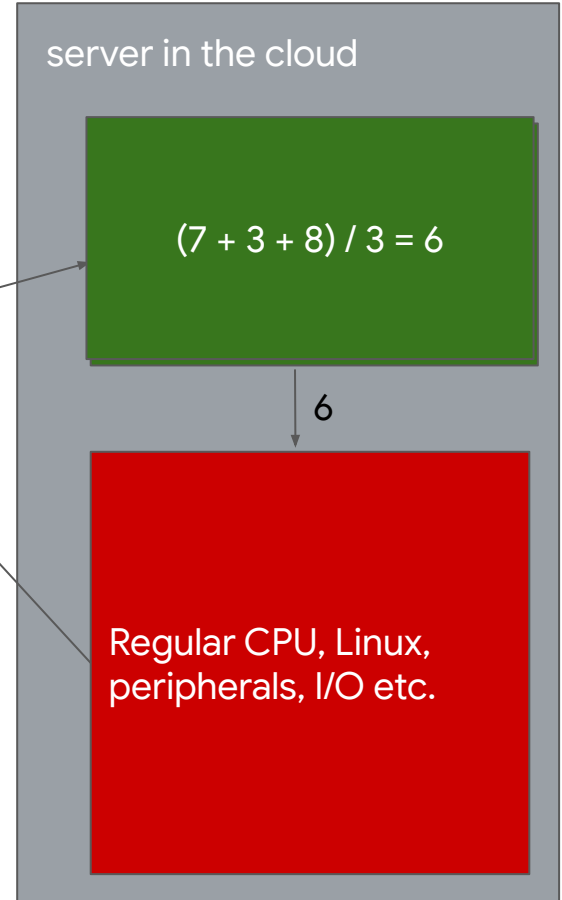
7  
3  
8

public network



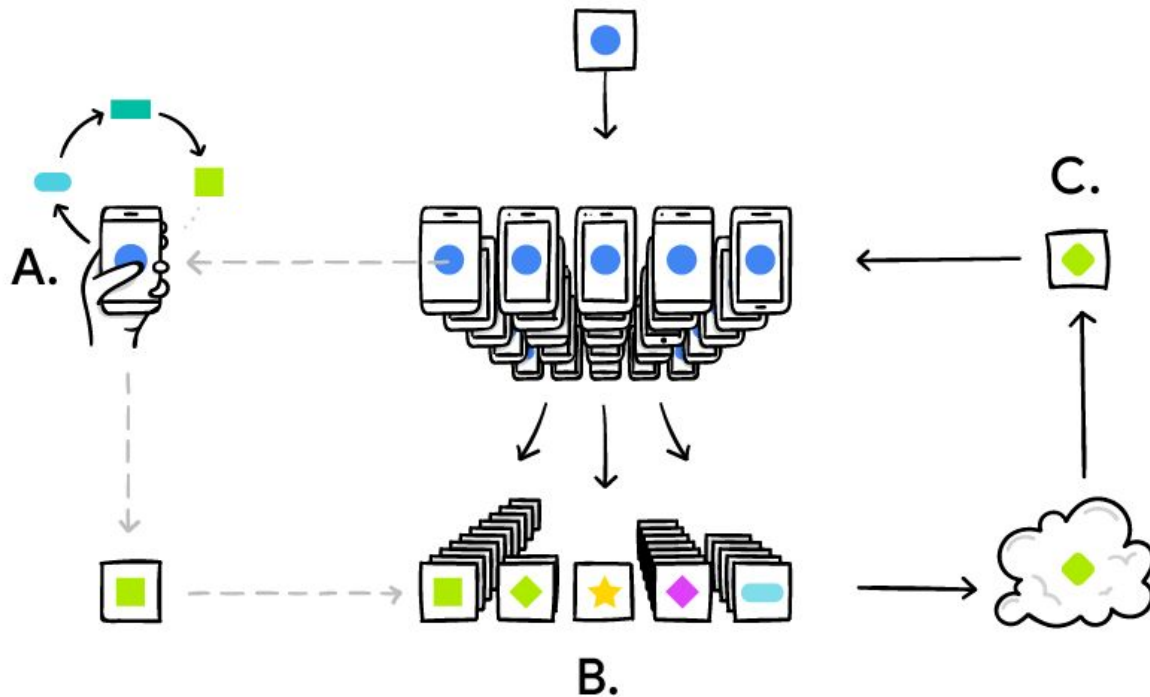
6

mobile device



**Policy:** outside the enclave only the average of the numbers seen can be observed, not the individual numbers.

# Federated Learning

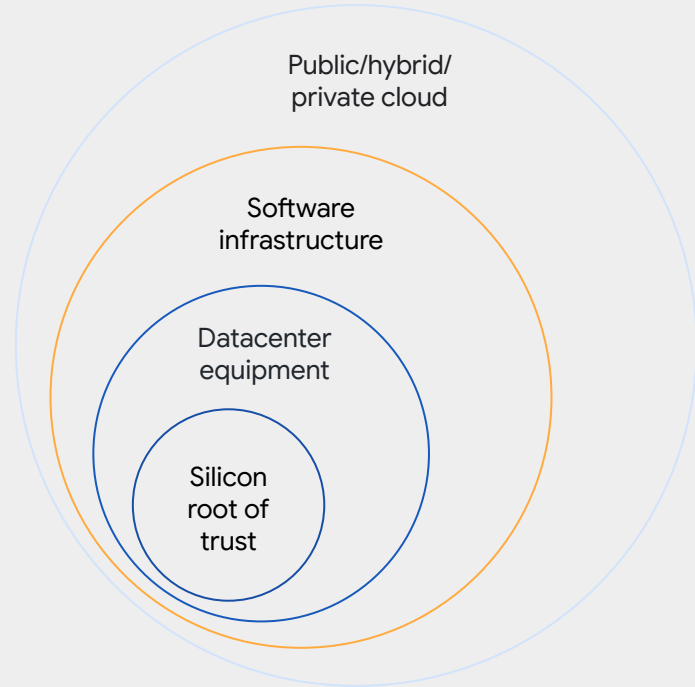




## A (Silicon) Root of Trust

The set of **inherently trusted** functions within a platform.

A silicon root of trust **is a chip**, below even the BIOS firmware, that provides those trusted functions.



# OpenTitan

More transparent, trustworthy,  
and secure RoT chip design

OpenTitan is **the first open source**  
silicon project building a transparent,  
high-quality reference design for  
silicon root of trust (RoT) chips.






master 2 branches 17 tags

Go to file

Add file

Code

 tjaychen [clkmgr] clkmgr markdown creation ...	✓ 375be34 5 hours ago	🕒 3,588 commits
 .github	Update CODEOWNERS	2 months ago
 ci	[ci] Use Ubuntu 18.04 in CI	23 days ago
 doc	[doc] Update D2 checklist and propagate updates to IPs	7 hours ago

README.md

# OpenTitan



## About the project

OpenTitan is an open source silicon Root of Trust (RoT) project. OpenTitan will make the silicon RoT design and implementation more transparent, trustworthy, and secure for enterprises, platform providers, and chip manufacturers. OpenTitan is administered by [lowRISC CIC](#) as a collaborative project to produce high quality, open IP for instantiation as a full-featured product. See the [OpenTitan site](#) and [OpenTitan docs](#) for more information about the project.

## About

OpenTitan: Open source silicon root of trust

[www.opentitan.org](https://www.opentitan.org)

[Readme](#)

[Apache-2.0 License](#)

# A radically different approach...

## Certified Programming with Dependent Types

A Pragmatic Introduction to the Coq Proof Assistant

Adam Chlipala



## A radically different approach...

- Specifications as dependently-typed programs in Coq/Gallina.
- Implementations as dependently-typed programs in Coq/Gallina.
- Proofs about relationship between specs and programs.
- Aggressive proof automation.
- Our specs: programs over lists representing streams of values for a singly-clocked synchronous circuit.
- Our implementation: extraction from Coq DSL to SystemVerilog.
- Verify “programs”, not “the compiler”

[POPL 2021 \(series\)](#) / [CPP 2021 \(series\)](#) / [Certified Programs and Proofs](#) /

## Lutsig: A Verified Verilog Compiler for Verified Circuit Development

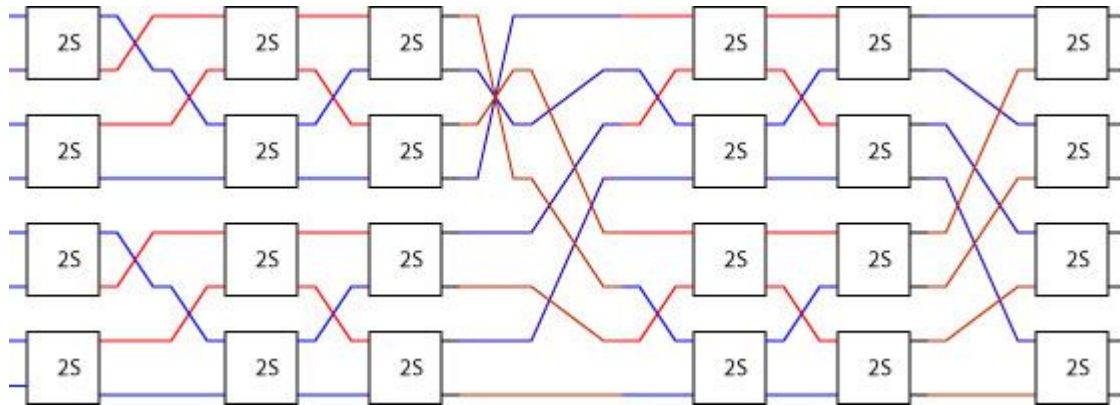
**Track** [CPP 2021](#)**When** **Tue 19 Jan 2021 07:15 - 07:30** at [CPP - Compilers and Interpreters](#) Chair(s): [Freek Wiedijk](#)

**Abstract** We report on a new verified Verilog compiler called Lutsig. Lutsig currently targets (a class of) FPGAs and is capable of producing technology mapped netlists for FPGAs. We have connected Lutsig to existing Verilog development tools, and in this paper we show how Lutsig, as a consequence of this connection, fits into a hardware development methodology for verified circuits in the HOL4 theorem prover. One important step in the methodology is transporting properties proved at the behavioral Verilog level down to technology mapped netlists, and Lutsig is the component in the methodology that enables such transportation.

**Link to Preprint** <http://www.cse.chalmers.se/~loow/papers/cpp2021.pdf>**Long presentation**

No Photo

Andreas Löw  
Chalmers University of Technology



```
bfly r 1 = r
```

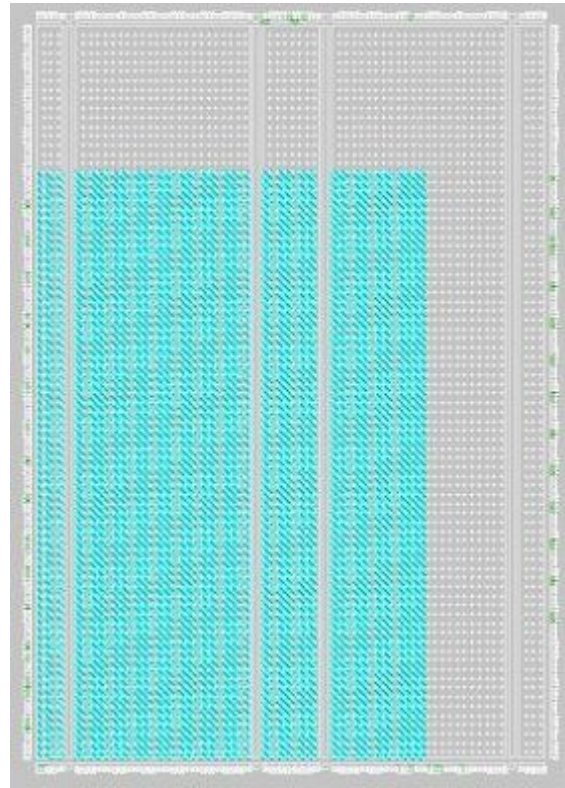
```
bfly r n = ilv (bfly r (n-1)) >-> evens r
```

```
sorter cmp 1 = cmp
```

```
sorter cmp n = two (sorter cmp (n-1)) >->
```

```
    sndList reverse >-> bfly cmp
```

```
n
```





**Cava = Coq + Lava**



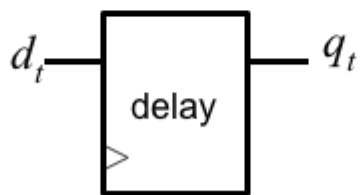
$$\forall t. o_t \Leftrightarrow \neg i_t$$



$$\forall t. c_t \Leftrightarrow a_t \wedge b_t$$



$$\forall t. c_t \Leftrightarrow a_t \vee b_t$$



$$\forall t. q_t \Leftrightarrow \text{if } t=0 \text{ then } 0 \text{ else } d_{t-1}$$

```
class MonadFix m => Lava m bit where
  inv :: bit -> m bit           -- inverter
  and2 :: (bit, bit) -> m bit  -- 2-input AND gate
  delay :: bit -> m bit       -- unit delay, with the initial output 0

-- An instance of the Lava class that performs circuit simulation.

instance Lava Identity [Bool] where
  inv x = return (map not x)
  and2 (as, bs) = return [a && b | (a, b) <- zip as bs]
  delay xs = return (False : xs)

{-
*Lava> runIdentity $ inv [False, True, True, False]
[True,False,False,True]
-}
```

```
data Component = INV Int Int
                | AND2 Int Int Int
                | DELAY Int Int
                deriving (Eq, Show)
```

```
data NetlistState = NetlistState Int [Component]
                  deriving (Eq, Show)
```

```
and2Gate :: (Int, Int) -> State NetlistState Int
and2Gate (i0, i1) = do NetlistState o components <- get
                       put (NetlistState (o+1) ((AND2 i0 i1 o):components))
                       return o
```

```
instance Lava (State NetlistState) Int where
  inv = invGate
  and2 = and2Gate
  delay = delayGate
```

```
-- An example of a nandGate composed with a left-to-right composition using a
-- Kleisli arrow.
```

```
nandGate :: Lava m bit => (bit, bit) -> m bit
nandGate = and2 >=> inv
```

```
sim3 :: [Bool]
sim3 = runIdentity $
  | | | | | nandGate ([False, True, False, True],
  | | | | | [False, False, True, True]
  | | | | | )
```

```
{-
*Lava> sim3
[True, True, True, False]
-}
```

```
-- Define a loop combinator to bend the select element of a pair to
-- pair circuit to allow us to express feedback loops for sequential circuits.
```



```
-}
```

```
loop :: MonadFix m => ((a, c) -> m (b, c)) -> a -> m b
```

```
loop circuit a
```

```
  = mdo (b, c) <- circuit (a, c)
```

```
    return b
```

```
-- Define a loop combinator to bend the select element of a pair to
-- pair circuit to allow us to express feedback loops for sequential circuits.
```



```
-}
```

```
loop :: MonadFix m => ((a, c) -> m (b, c)) -> a -> m b
```

```
loop circuit a
```

```
  = mdo (b, c) <- circuit (a, c)
```

```
    return b
```

```
Class Cava (signal : SignalType -> Type) := {
  cava : Type -> Type;
  monad :> Monad cava;
  (* Constant values. *)
  constant : bool -> signal Bit;
  constantV : forall {A} {n : nat}, Vector.t (signal A) n -> signal (Vec A n);
  (* Default values. *)
  defaultSignal : forall {t: SignalType}, signal t;
  (* SystemVerilog primitive gates *)
  inv : signal Bit -> cava (signal Bit);
  and2 : signal Bit * signal Bit -> cava (signal Bit);
  nand2 : signal Bit * signal Bit -> cava (signal Bit);
  or2 : signal Bit * signal Bit -> cava (signal Bit);
  nor2 : signal Bit * signal Bit -> cava (signal Bit);
  xor2 : signal Bit * signal Bit -> cava (signal Bit);
  xnor2 : signal Bit * signal Bit -> cava (signal Bit);
```



## Section WithCava.

```
Context `{semantics:Cava}.
```

```
Inductive Circuit : Type -> Type -> Type :=
```

```
| Comb : forall {i o}, (i -> cava o) -> Circuit i o
```

```
| Compose : forall {i t o}, Circuit i t -> Circuit t o -> Circuit i o
```

```
| First : forall {i o t}, Circuit i o -> Circuit (i * t) (o * t)
```

```
| Second : forall {i o t}, Circuit i o -> Circuit (t * i) (t * o)
```

```
| LoopInitCE :
```

```
  forall {i o : Type} {s : SignalType} (resetval : combType s),
```

```
    Circuit (i * signal s) (o * signal s) ->
```

```
    Circuit (i * signal Bit) o
```

```
| DelayInitCE :
```

```
  forall {t} (resetval : combType t),
```

```
    Circuit (signal t * signal Bit) (signal t)
```

```
.
```

## Section WithCava.

```
Context {signal} {semantics: Cava signal}.
```

```
Existing Instance monad. (* make sure cava Monad instance takes precedence *)
```

```
(*****)
```

```
(* Forks in wires *)
```

```
(*****)
```

```
(* forks a wire into two *)
```

```
Definition fork2 {A} (a:A) := ret (a, a).
```

```
(*****)
```

```
(* Operations over pairs. *)
```

```
(*****)
```

```
(* applies f to the first element of a pair *)
```

```
Definition first {A B C} (f : A -> cava C) (ab : A * B) : cava (C * B) :=
```

```
  let '(a, b) := ab in
```

```
  c <- f a ;;
```

```
  ret (c, b).
```

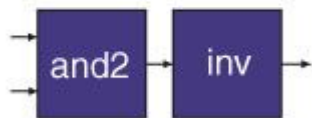
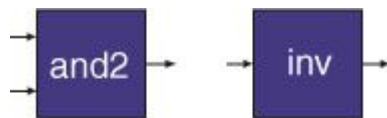
```
(* n should be the maximum depth of the tree, inputs must not be empty *)
```

```
Fixpoint tree_generic {T: Type}
  (circuit: T -> T -> cava T) (error: T)
  (n : nat) (inputs : list T) : cava T :=
  match n with
  | 0 =>
    (* for a depth of 0, only a singleton list is possible *)
    match inputs with
    | [t]%list => ret t
    | _ => ret error (* should not get here *)
    end
  | S n' =>
    if (1 <? length inputs)
    then
      (* if there are at least 2 elements, halve the input list *)
      let mid := (length inputs) / 2 in
      let iL := firstn mid inputs in
      let iR := skipn mid inputs in
      aS <- tree_generic circuit error n' iL ;;
      bS <- tree_generic circuit error n' iR ;;
      circuit aS bS
    else
      (* same as 0-depth case -- only a singleton list is possible *)
      match inputs with
      | [t]%list => ret t
      | _ => ret error (* should not get here *)
      end
    end.
end.
```

```
Definition tree {t : SignalType}
```

```
  (circuit: signal t * signal t -> cava (signal t))
  {n} (v : signal (Vec t n)) : cava (signal t) :=
  v <- unpackV v ;;
  tree_generic (fun a b => circuit (a,b))
  defaultSignal (Nat.log2_up n) (to_list v).
```

**Definition** `nand2_gate := and2 ==> inv.`



Kleisli arrow

**Definition** `nand2_gate_alt `(a, b) : m (signal bit) :=  
x <- and2 (a, b) ;;  
y <- inv x ;;  
ret y.`

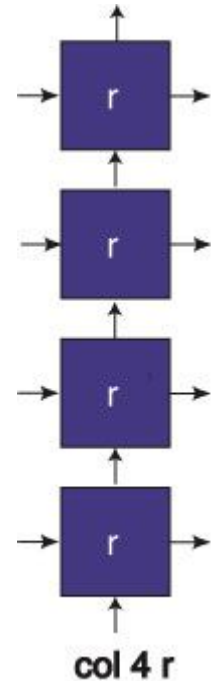
```
Definition halfAdder '(a, b) :=  
  partial_sum <- xor2 (a, b) ;;  
  carry <- and2 (a, b) ;;  
  ret (partial_sum, carry).
```

```
Definition fullAdder '(cin, (a, b))  
  : cava (signal Bit * signal Bit) :=  
  '(abl, abh) <- halfAdder (a, b) ;;  
  '(abcl, abch) <- halfAdder (abl, cin) ;;  
  cout <- or2 (abh, abch) ;;  
  ret (abcl, cout).
```

```

(* Unsigned adder for n-bit vectors with carry bits both in and out *)
Definition addC {n : nat}
  (inputs : signal (Vec Bit n) * signal (Vec Bit n) * signal Bit) :
  cava (signal (Vec Bit n) * signal Bit) :=
  let '(x, y, cin) := inputs in
  x <- unpackV x ;;
  y <- unpackV y ;;
  col fullAdder cin (vcombine x y).

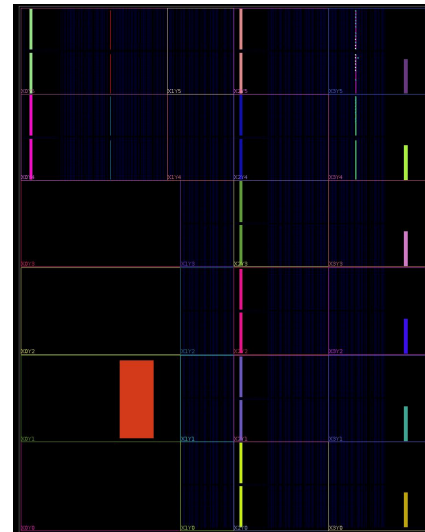
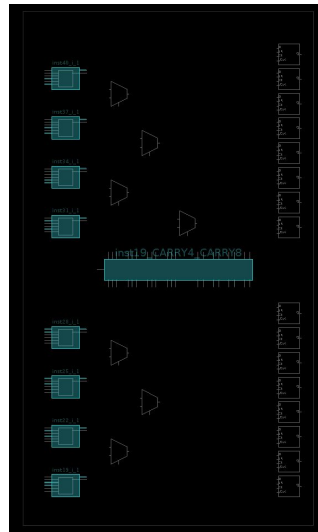
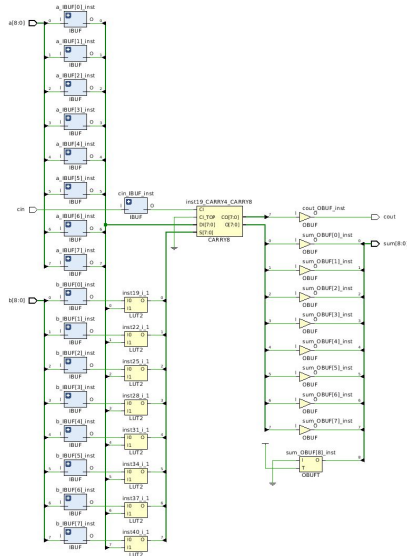
```



# An ripple-carry adder

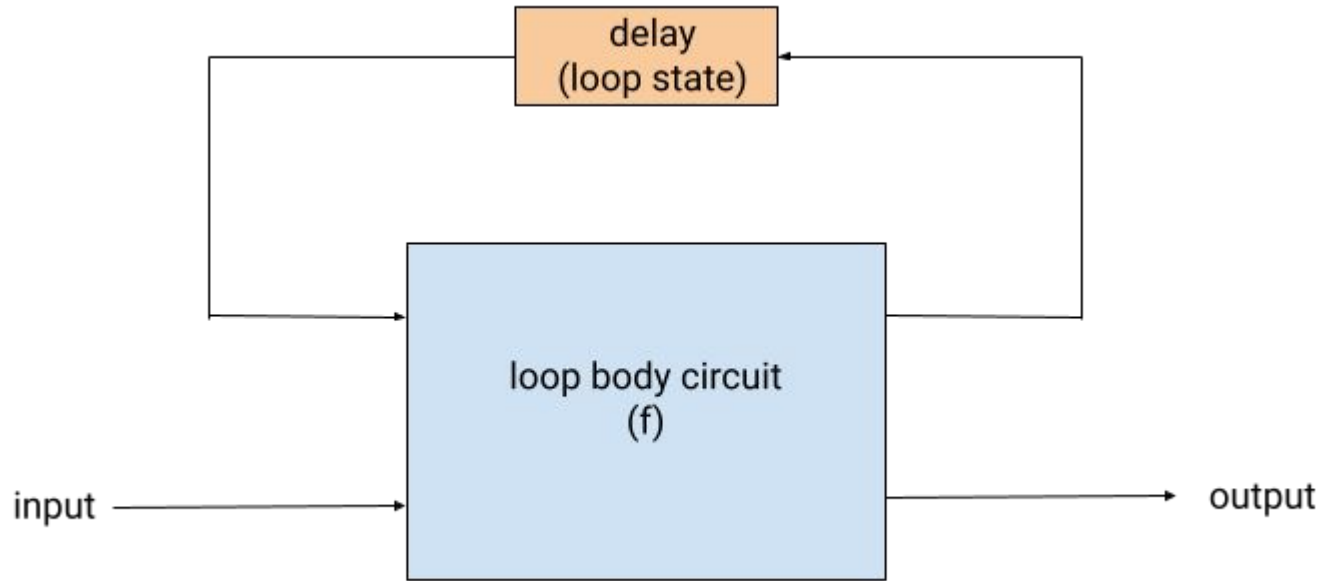
```

Definition unsignedAdderV {n : nat}
  (inputs: signal Bit * (Vector.t (signal Bit * signal Bit)) n) :
  m (Vector.t (signal Bit) n * signal Bit) :=
  colV fullAdder inputs.
  
```



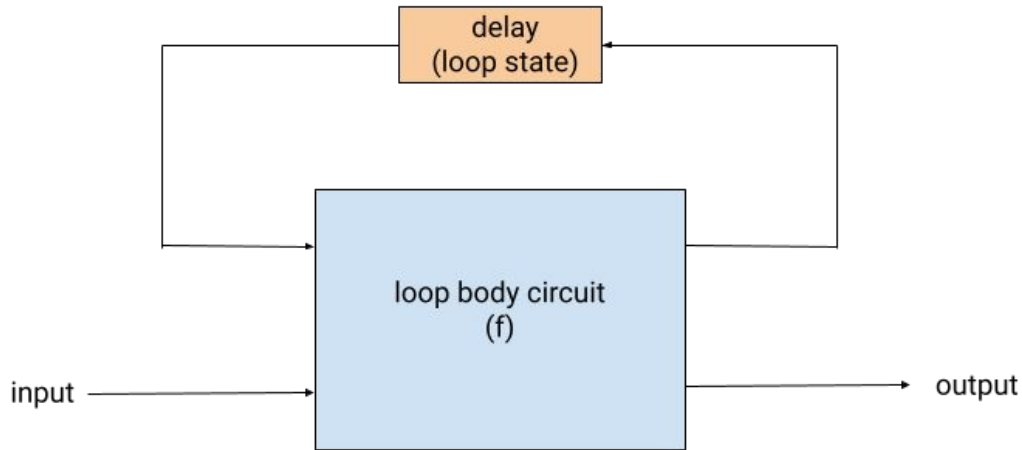






**Definition** `sum` {n : nat}

```
: Circuit (signal (Vec Bit n)) (signal (Vec Bit n)) :=  
Loop (Comb (addN ==> fork2)).
```



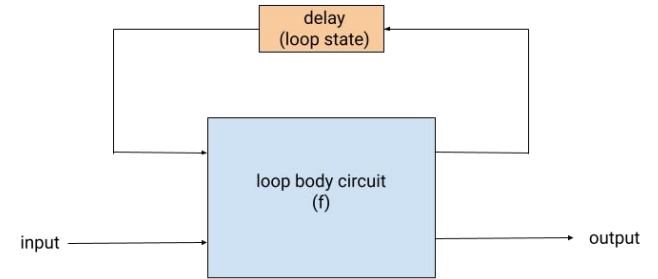
```
Definition sum8Netlist := makeCircuitNetlist (sum_interface (n:=8)) sum.  
Definition sum8_tb_inputs := map (N2Bv_sized 8) [3; 5; 7; 2; 4; 6].  
Definition sum8_tb_expected_outputs := simulate sum sum8_tb_inputs.  
  
(* print out the expected outputs according to the Coq semantics *)  
Compute map Bv2N (sum8_tb_expected_outputs).
```

```
= [3; 8; 15; 17; 21; 27]  
: list N
```

```
clang++ -L/usr/local/opt/sqlite/lib sum8_tb.o verilated.o  
obj_dir/Vsum8_tb
```

```
10: tick = 0, i = 3, o = 3  
20: tick = 1, i = 5, o = 8  
30: tick = 2, i = 7, o = 15  
40: tick = 3, i = 2, o = 17  
50: tick = 4, i = 4, o = 21  
60: tick = 5, i = 6, o = 27
```

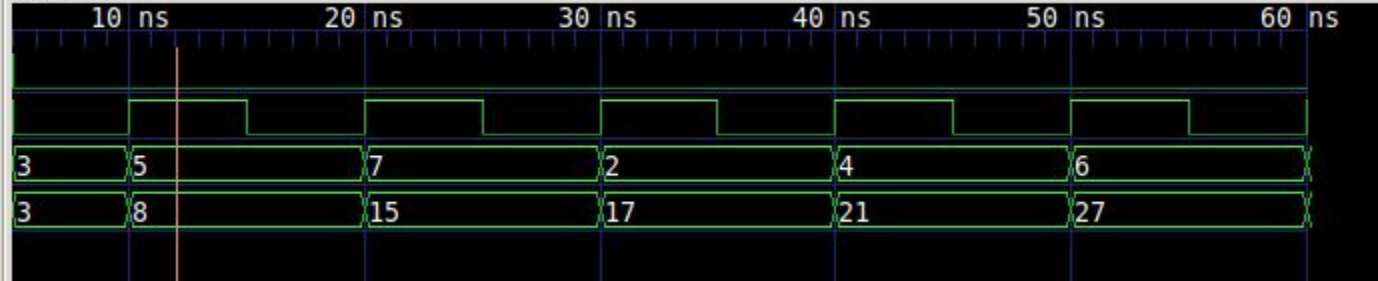
PASSED



Signals

Time  
rst=0  
clk=1  
i[7:0]=5  
o[7:0]=8

Waves



```
// Cava auto-generated SystemVerilog. Do not hand edit.
```

```
module xadder_tree32_8(
```

```
    input  logic[7:0] inputs[32],
```

```
    output logic[7:0] sum
```

```
);
```

```
timeunit 1ns; timeprecision 1ns;
```

```
logic zero;
```

```
logic one;
```

```
logic[743:0] net;
```

```
// Constant nets
```

```
assign zero = 1'b0;
```

```
assign one = 1'b1;
```

```
assign sum = {net[742],net[739],net[736],net[733],net[730],net[727],net[724],net[721]};
```

```
MUXCY inst_1 (.O(net[743]),.S(net[741]),.CI(net[740]),.DI(net[358]));
```

```
XORCY inst_2 (.O(net[742]),.CI(net[741]),.LI(net[740]));
```

```
xor inst_3 (net[741],net[358],net[718]);
```

```
MUXCY inst_4 (.O(net[740]),.S(net[738]),.CI(net[737]),.DI(net[355]));
```

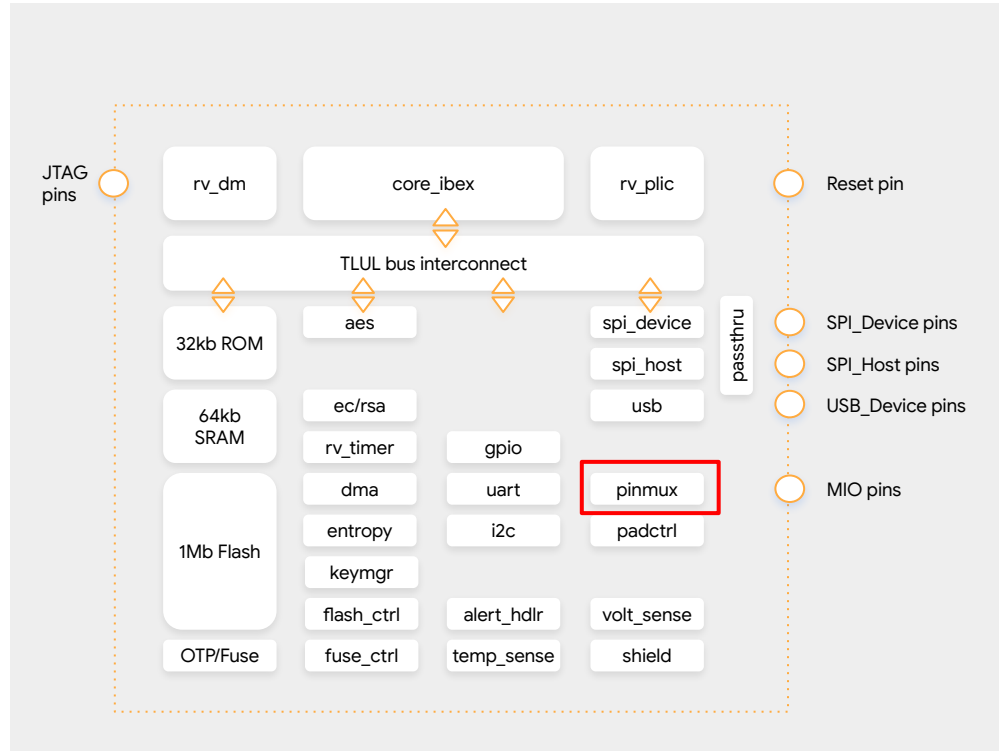
```
XORCY inst_5 (.O(net[739]),.CI(net[738]),.LI(net[737]));
```

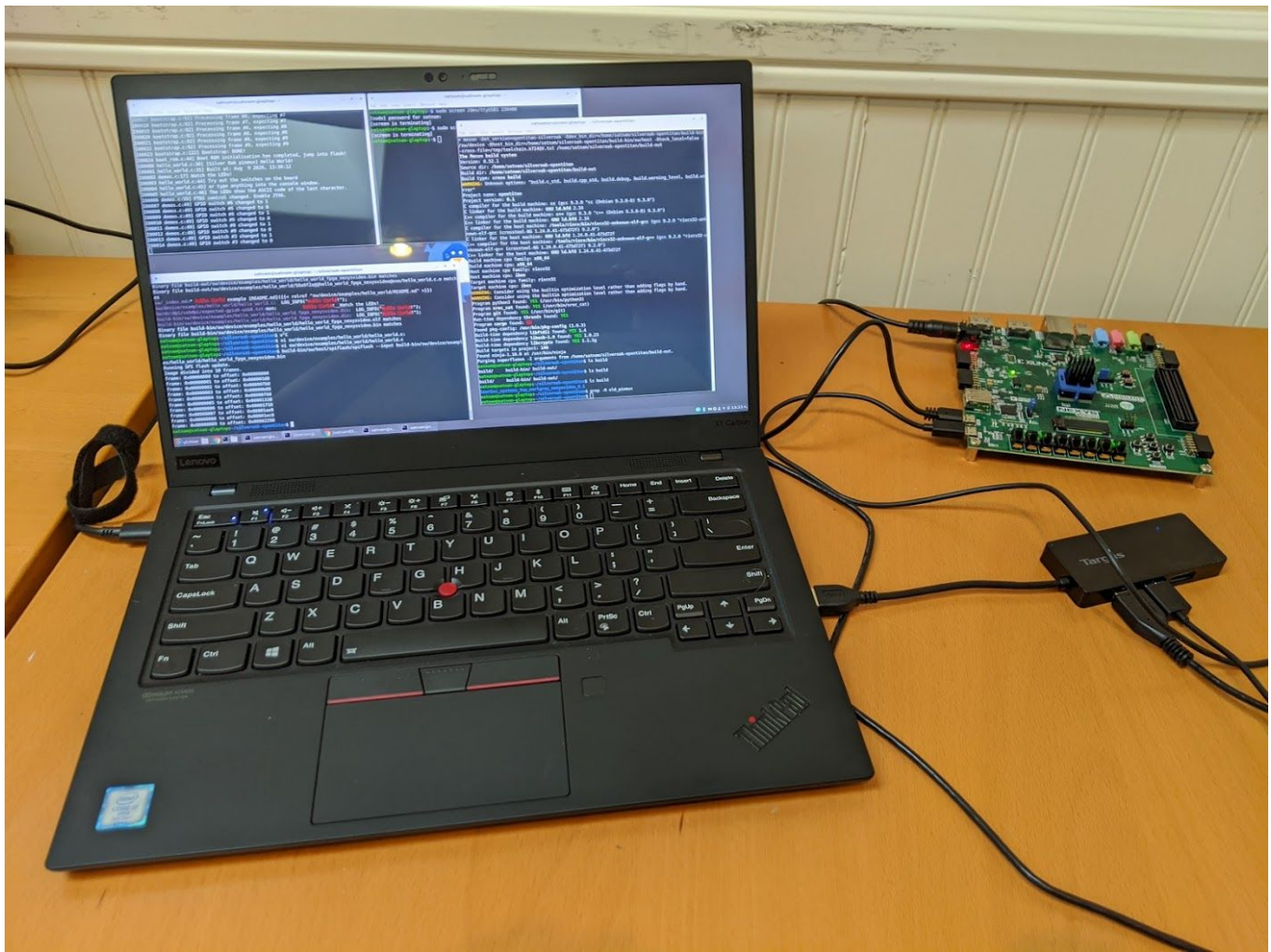
```
xor inst_6 (net[738],net[355],net[715]);
```

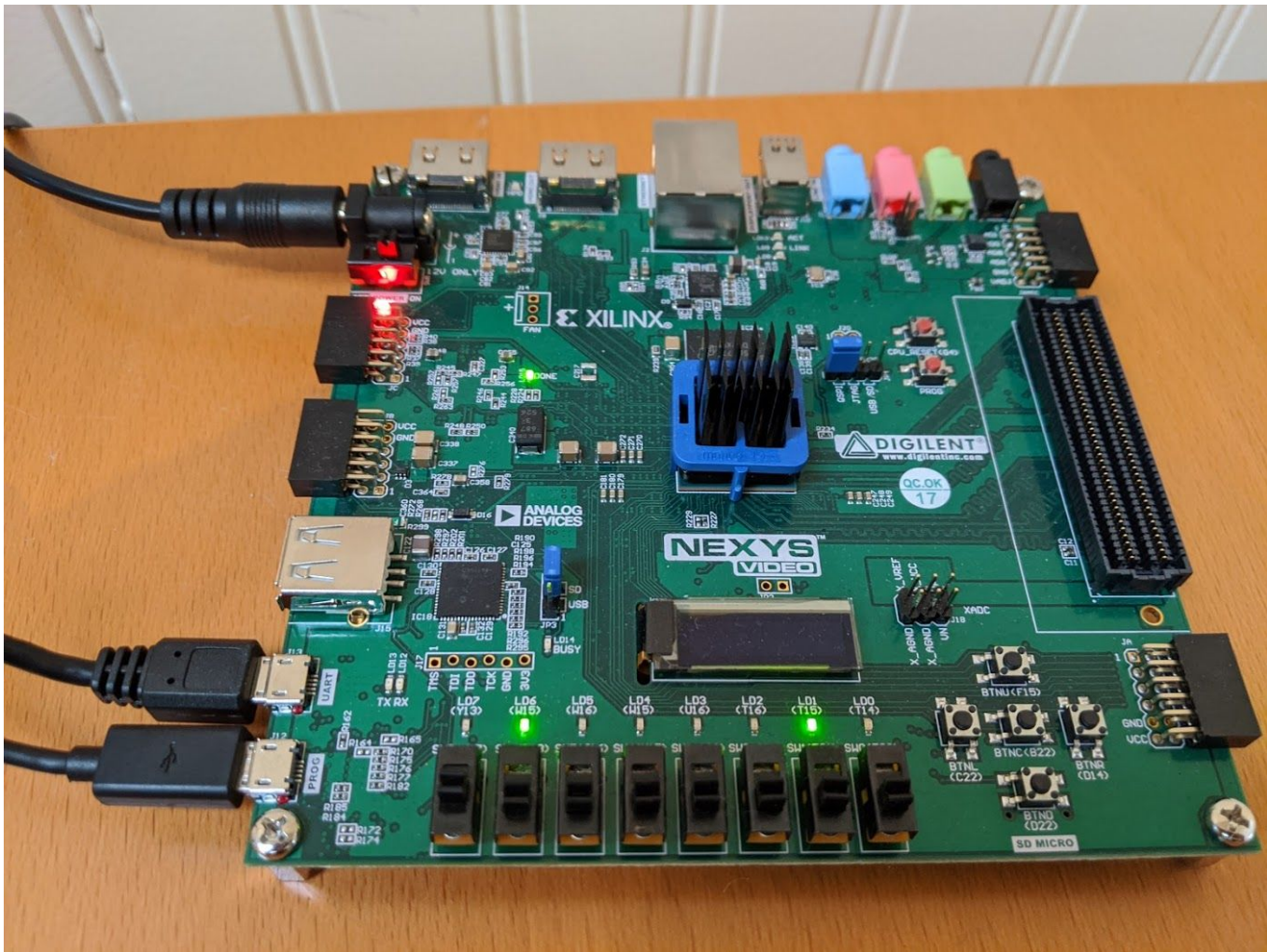
```
MUXCY inst_7 (.O(net[737]),.S(net[735]),.CI(net[734]),.DI(net[352]));
```

```
...
```

# Generation of SystemVerilog from Cava that integrates into an existing system







lowrisc\_systems\_top\_earlgrey\_nexysvideo\_0.1 - [/home/satnam/opentitan/build/lowrisc\_systems\_top\_earlgrey\_nexysvideo\_0.1/synth-vivado/lowrisc\_systems\_top\_earlgrey\_nexysvideo\_0.1.xpr] - Vivad... - Implementation Complete

File Edit Flow Tools Reports Window Layout View Help Quick Access

Flow Navigator PROJECT MANAGER

- Settings
- Add Sources
- Language Templates
- IP Catalog

IP INTEGRATOR

- Create Block Design
- Open Block Design
- Generate Block Design

SIMULATION

- Run Simulation

RTL ANALYSIS

- Open Elaborated Design

SYNTHESIS

- Run Synthesis
- Open Synthesized Design
  - Constraints Wizard
  - Edit Timing Constraints
  - Set Up Debug
  - Report Timing Summary
  - Report Clock Networks
  - Report Clock Interaction
  - Report Methodology
  - Report DRC
  - Report Noise
  - Report Utilization
  - Report Power
  - Schematic

IMPLEMENTATION

- Run Implementation
- Open Implemented Design

IMPLEMENTED DESIGN - xc7a200tcbg484-1 (active)

Sources Netlist

- top\_earlgrey\_nexysvideo
  - Nets (107)
  - Leaf Cells (35)
    - clo\_jtag\_tck\_p2d\_BUF0\_inst (BUF0)
    - IO\_DPS0\_IBUF\_inst (IBUF)
    - IO\_DPS1\_IBUF\_inst (IBUF)
    - IO\_DPS2\_OBUF\_inst (OBUF)
    - IO\_DPS2\_OBUFT\_inst\_1\_4 (LUT2)
    - IO\_DPS3\_IBUF\_inst (IBUF)
    - IO\_DPS4\_IBUF\_inst (IBUF)
    - IO\_DPS5\_IBUF\_inst (IBUF)
    - IO\_DPS6\_IBUF\_inst (IBUF)
    - IO\_DPS7\_IBUF\_inst (IBUF)
    - IO\_GPO\_IOBUF\_inst (IOBUF)

Cell Properties

IO\_DPS5\_IBUF\_inst

Name: IO\_DPS5\_IBUF\_inst

Reference name: IBUF

Type: IO

BEL: INBUF\_EN  Fix

Site: P16

Tile: LIOB33\_X0Y101

Clock region: X0Y2

Number of cell pins: 2

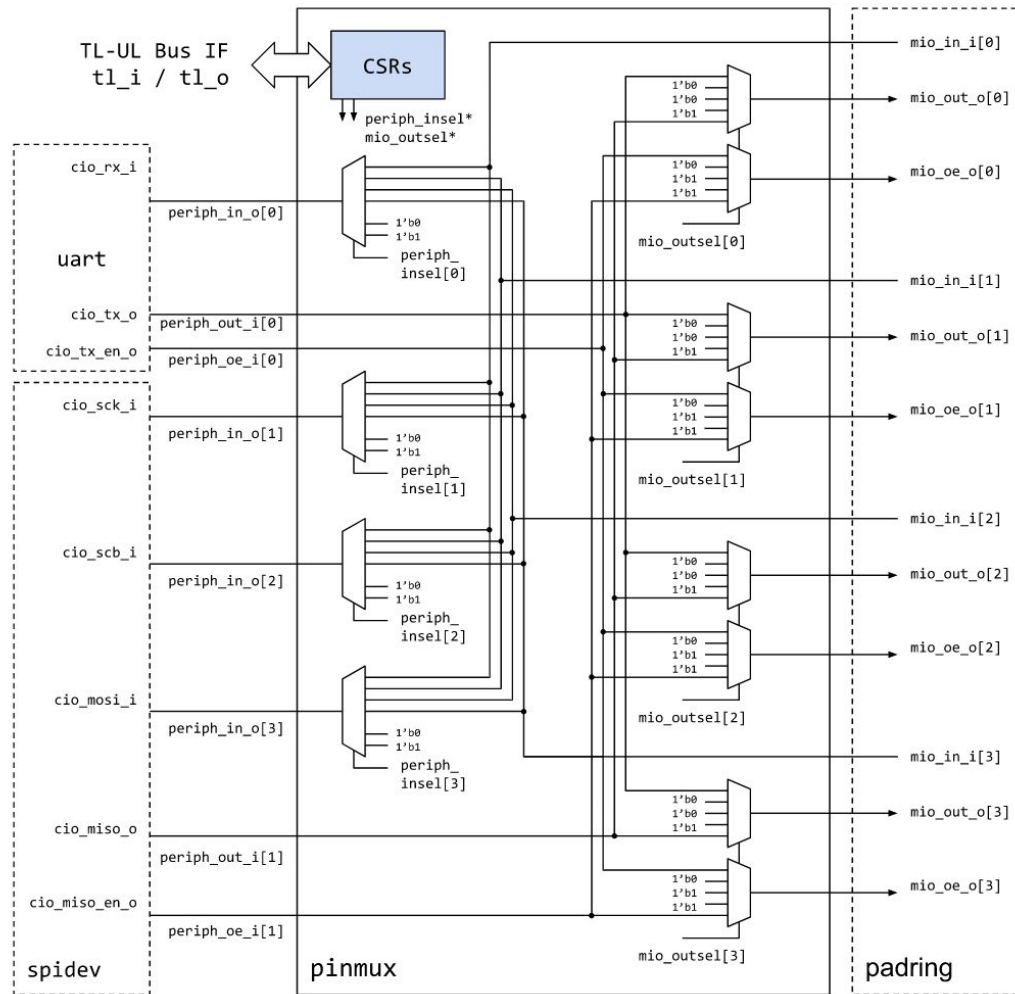
General Properties Nets Ce

Project Summary Device

Tcl Console Messages Log Reports Design Runs Power DRC Methodology Timing

Report	Report Type	Options	Modified	Size
<ul style="list-style-type: none"> <li>Synthesis               <ul style="list-style-type: none"> <li>Synth Design (synth_design)                   <ul style="list-style-type: none"> <li>synth_1_synth_report_utilization_0</li> <li>synth_1_synth_synthesis_report_0</li> </ul> </li> </ul> </li> </ul>	Report on utilization of resources on the targeted device (report_utilization)		8/9/20, 8:38 PM	8.0 KB
<ul style="list-style-type: none"> <li>Implementation               <ul style="list-style-type: none"> <li>impl_1                   <ul style="list-style-type: none"> <li>Design Initialization (init_design)</li> </ul> </li> </ul> </li> </ul>	Vivado Synthesis Report		8/9/20, 8:38 PM	839.3 KB





```

//////////
// Input Mux //
//////////

for (genvar k = 0; k < pinmux_reg_pkg::NPeriphIn; k++) begin : gen_periph_in
    logic [2**$clog2(pinmux_reg_pkg::NMioPads+2)-1:0] data_mux;
    // stack input and default signals for convenient indexing below
    // possible defaults: constant 0 or 1
    assign data_mux = $bits(data_mux)'({mio_in_i, 1'b1, 1'b0});
    // index using configured insel
    assign mio_to_periph_o[k] = data_mux[reg2hw.periph_insel[k].q];
end

```

```

(* Input Mux *)
let data_in_mux := VecLit ([const0; const1] ++ mio_in_i) in
let mio_to_periph_o :=
    Vector.map (fun k => IndexAt data_in_mux (kq "periph_insel" reg2hw k))
        (vseq 0 NPeriphIn) in

```

# Pinmux re-implementation

Resource	Utilization	Available	Utilization %
LUT	1286	133800	0.96
FF	430	267600	0.16
IO	318	400	79.50
BUFG	1	32	3.13

Only pinmux utilization (original design)

Resource	Utilization	Available	Utilization %
LUT	1226	133800	0.92
FF	430	267600	0.16
IO	318	400	79.50
BUFG	1	32	3.13

Cava generated (from Coq) Silver Oak pinmux

**Passes formal equivalenace check (LEC)**

# Likewise for AES subcomponents: aes\_sub\_bytes

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	1152	0	134600	0.86
LUT as Logic	1152	0	134600	0.86
LUT as Memory	0	0	46200	0.00
Slice Registers	0	0	269200	0.00
Register as Flip Flop	0	0	269200	0.00
Register as Latch	0	0	269200	0.00
F7 Muxes	512	0	67300	0.76
F8 Muxes	0	0	33650	0.00

Original SystemVerilog aes\_sub\_bytes

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	1152	0	134600	0.86
LUT as Logic	1152	0	134600	0.86
LUT as Memory	0	0	46200	0.00
Slice Registers	0	0	269200	0.00
Register as Flip Flop	0	0	269200	0.00
Register as Latch	0	0	269200	0.00
F7 Muxes	512	0	67300	0.76
F8 Muxes	0	0	33650	0.00

Cava version from Coq implementation with formal specification and proof

# AES spec : Coq vs FIPS

```

Definition cipher (first_key last_key : key)
  (middle_keys : list key) (input : state) : state :=
  let st := input in
  let st := add_round_key st first_key in

  let st := fold_left
    (fun (st : state) (round_key : key) =>
      let st := sub_bytes st in
      let st := shift_rows st in
      let st := mix_columns st in
      let st := add_round_key st round_key in
      st) middle_keys st in
  st.

```

```

Cipher(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])
begin
  byte state[4,Nb]
  state = in
  AddRoundKey(state, w[0, Nb-1])

  for round = 1 step 1 to Nr-1
    SubBytes(state)
    ShiftRows(state)
    MixColumns(state)
    AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])
  end for

  SubBytes(state)
  ShiftRows(state)
  AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1])
  out = state
end

```

**Definition** cipher\_round

```
(is_decrypt : signal Bit) (input: signal state) (key : signal key)
: cava (signal state) :=
(sub_bytes is_decrypt ==>
 shift_rows is_decrypt ==>
 mix_columns is_decrypt ==>
 add_round_key key) input.
```

## Definition cipher\_step

```

(is_decrypt : signal Bit) (* called op_i in OpenTitan *)
(key_rcon_data : signal key * signal round_constant * signal state)
(round_i : signal round_index)
: cava (signal key * signal round_constant * signal state) :=
let '(round_key, rcon, input) := key_rcon_data in
is_first_round <- round_i ==? round_0 ;;
is_final_round <- round_i ==? num_rounds_regular ;;
(* add_round_key_in_sel :
  1 if round_i = 0, 2 if round_i = num_rounds_regular, 0 otherwise *)
let add_round_key_in_sel := unpeel [is_first_round; is_final_round]%vector in
is_middle_round <- nor2 (is_first_round, is_final_round) ;;
(* round_key_sel : 1 for a decryption middle round, 0 otherwise *)
round_key_sel <- and2 (is_middle_round, is_decrypt) ;;
key_expand_and_round is_decrypt (round_key, rcon, input)
add_round_key_in_sel round_key_sel round_i.

```

## Definition cipher

```

(is_decrypt : signal Bit) (* called op_i in OpenTitan *)
(initial_key : signal key) (initial_rcon : signal round_constant)
(round_indices : list (signal round_index)) (input : signal state)
: cava (signal state) :=
'(_, _, out) <- foldLM (cipher_step is_decrypt) round_indices
(initial_key, initial_rcon, input) ;;
ret out.

```

# Equivalent inverse cipher implements inverse

Lemma inverse\_cipher\_id :

```
forall first_key last_key middle_keys block,  
  equivalent_inverse_cipher  
    last_key first_key (map inv_mix_columns_key (rev middle_keys))  
    (cipher first_key last_key middle_keys block) = block.
```

*Proof statement*

Proof.

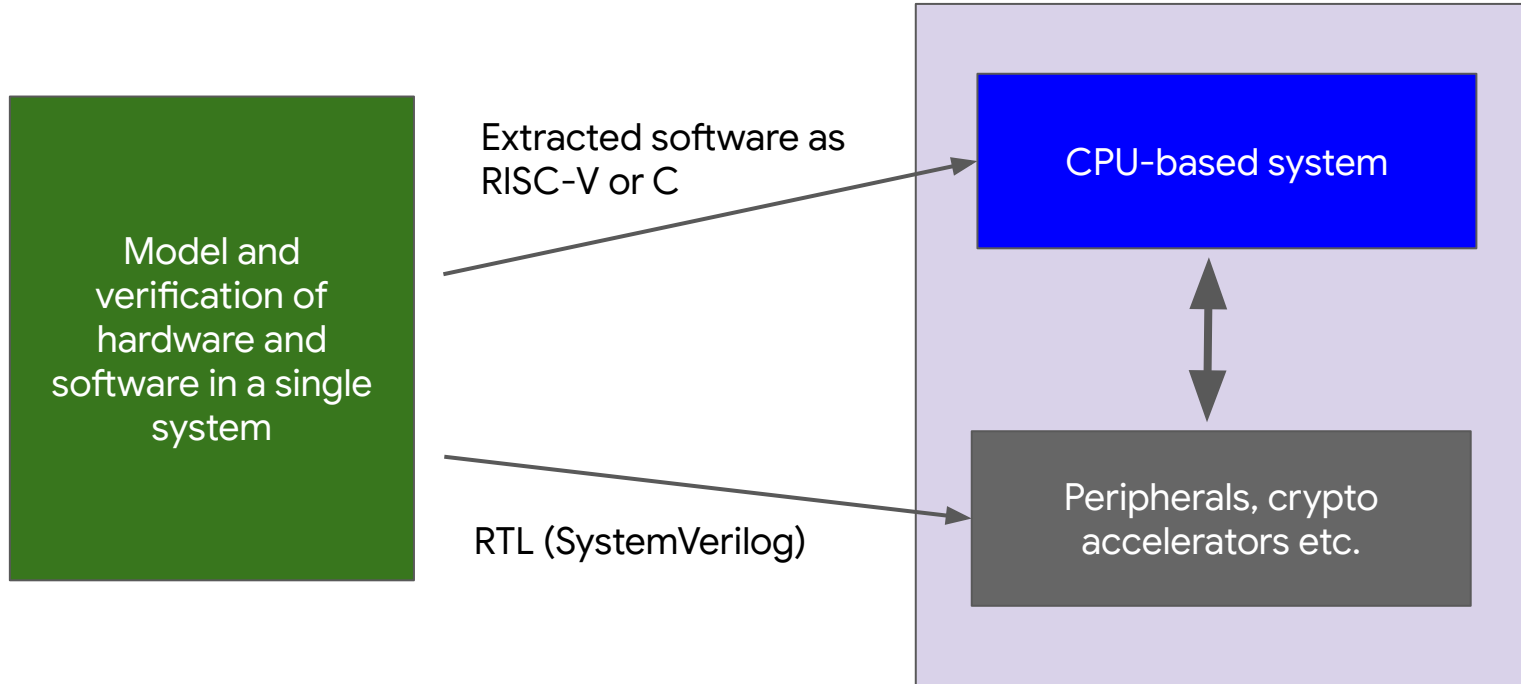
```
cbv [cipher equivalent_inverse_cipher].  
apply add_round_key_cancel. revert first_key block.  
induction middle_keys; intros; listsimpl.  
{ repeat t. }  
{ rewrite IHmiddle_keys. repeat t. }
```

Qed.

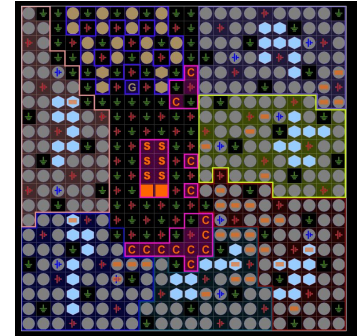
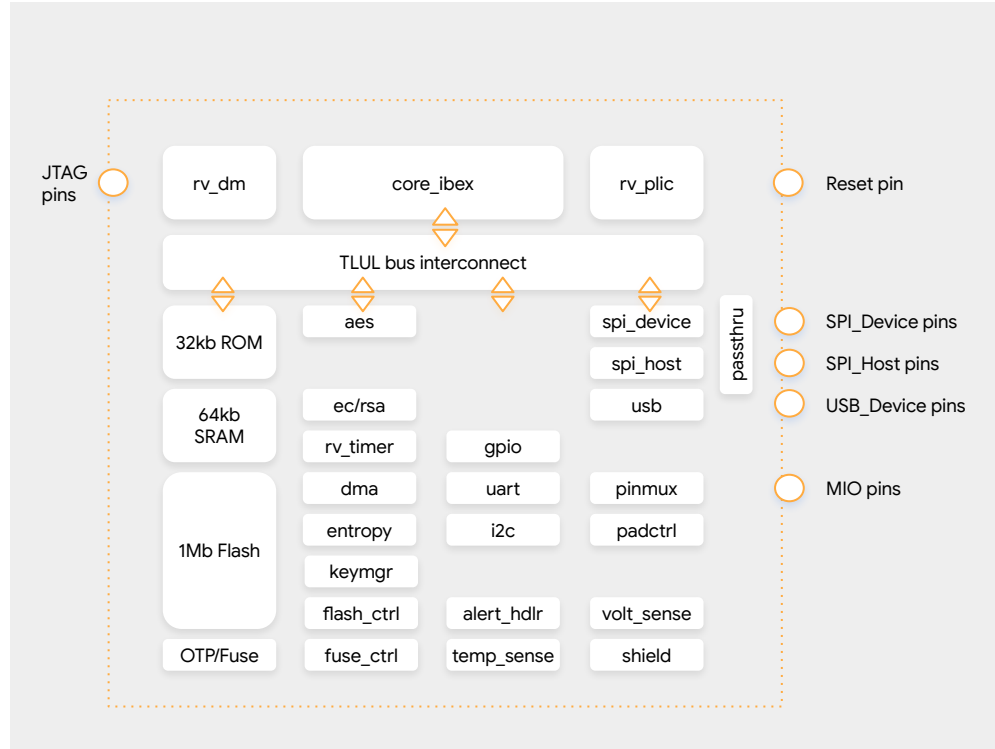
*Proof body*



# Co-Design of hardware *and* software



# Formally verified software **and** hardware extracted from Coq



<https://github.com/project-oak/oak-hardware>

# PLDI 2021 Lightbulb paper from MIT

## Integration Verification across Software and Hardware for a Simple Embedded System

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

### 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 against the same formal model of the interface between them can be used to shake out these bugs. Doing so for every interface in the system can, in principle, yield unparalleled assurance of the correctness and security of the system as a whole. However, there have been remarkably few efforts that carry out this exercise, and all of them have simplified the task by restricting interactivity of the application, inventing new simplified instruction sets, and using unrealistic input and output mechanisms. We report on the first verification of a realistic embedded system, with its application software, device drivers, compiler, and RISC-V processor represented inside the Coq proof assistant as one mathematical object, with a machine-checked proof of functional correctness. A key challenge is structuring the proof modularly, so that further refinement of the components or expansion of the system can proceed without revisiting the rest of the system.

### ACM Reference Format:

Andres Erbsen, Samuel Gruetter, Joonwon Choi, Clark Wood, and Adam Chlipala. 2021. 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 '21)*, June 20–25, 2021, Virtual, Canada. ACM, New York, NY, USA, 16 pages. <https://doi.org/10.1145/3453483.3454065>

### 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 that remain in scaling up the scope and realism of verification and reducing the effort required. Our development of an Ethernet-connected IoT lightbulb controller culminates in a single Coq proof relating the network packets entering our integrated system through memory-mapped I/O (MMIO) to the action the controller takes by emitting MMIO writes, ruling out any bugs or vulnerabilities that could be exploited over the network. In particular, the proof spans a pipelined

# PLDI 2021 Lightbulb paper from MIT

Key:

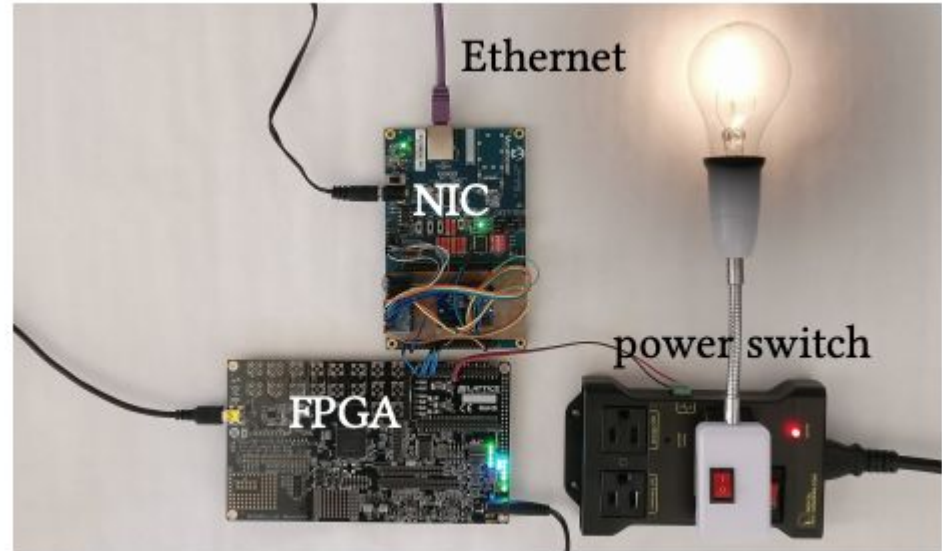
✓ met

~ partially met

✗ not met

- not applicable

	seL4 [23]	VST+CertikOS [30]	CompCertMC [41]	Everest [6]	Serval [33]	Vigor [43]	CLI stack [5]	Verisoft [2]	CakeML [25, 29]	This paper
Applications										
OS and/or drivers	█	█		█	█	█	█	█	█	█
Source language			█							
Assembly				█						
Machine code	█		█		█	█	█	█	█	█
HDL							█	█	█	█
Integration verification	~	~	✓	✗	✓	✓	✓	✓	✓	✓
One proof assistant	✓	✓	✓	✗	✗	✗	✓	✓	✓	✓
Modularity	~	✓	✓	~	✗	✓	✓	✓	✓	✓
Standardized ISA	✓	✓	✓	✓	✓	✓	~	✗	✗	✓
HW optimizations	-	-	-	-	-	-	~	✓	✗	✓
Realistic I/O	✓	~	✗	✗	~	✓	✗	~	✗	✓



## HW/SW Co-design in a single model

- Firmware written in bedrock2 from MIT
- RISC-V code generated from bedrock2, with semantics in Coq
- Hardware peripherals in Coq, produced using the Cava hardware DSL
- Model HW/Sw interface via memory-mapped I/O and TileLink bus-interface
- AES, UART and other OpenTitan peripherals

## Fragment of AES OpenTitan firmware (C vs. bedrock2)

```

void aes_iv_put(const void *iv) {
    // Write the four initialization vector registers.
    for (int i = 0; i < AES_NUM_REGS_IV; ++i) {
        REG32(AES_IV0(0) + i * sizeof(uint32_t)) = ((uint32_t *)iv)[i];
    }
}

```

```

Definition aes_iv_put : func :=
  let iv := "iv" in
  let i := "i" in
  ("b2_iv_put",
   ([AES_IV0; AES_NUM_REGS_IV; iv], [], bedrock_func_body:(
     i = 0 ;
     while (i < AES_NUM_REGS_IV) {
       output! WRITE (AES_IV0 + (i * 4), load4( iv + (i * 4) ));
       i = i + 1
     }
   )))

```

## Status

- **Formal specification in Coq, Cava implementation in Coq, proof in Coq for AES hardware peripheral done (but not yet for masked-write version).**
- **Extracted SystemVerilog passes all OpenTitan simulation tests (Verilator).**
- **Circuit synthesizes and implements in Xilinx FPGAs tools to produce drop-in replacement of the same size and speed as original.**
- **Drop-in replacement circuits works on FPGA as drop-in replacement.**
- **Now tackling firmware i.e. aes.c as well as aes.sv**
- **Weakness: control (esp. for TileLink bus protocol)**

# Precursor





## Long term goal: a secure communication device

- A “blueprint” for a secure communication device, design downloadable from GitHub.
- A secure core based on OpenTitan, with cycle-by-cycle semantics in Coq based on Cava and bedrock2
- A screen and keyboard, high firmware (bedrock2) and hardware (Cava) for UART, IC2, USB etc.
- An “application” layer based on tock OS on top of OpenTitan that provides UI, wireless etc. and a lower level of assurance, which runs Oak policies for the secure processing and communication of private data.
- Same high assurance nucleus can be used for many other IoT applications