Coquet: A Coq library for verifying hardware

Thomas Braibant

Inria Rhône-Alpes - Université Joseph Fourier - LIG

Octobre 2011

Representing circuits with predicates (or functions).

- Some definitions:

$$Xor(i_1, i_2, o) \triangleq (o = \neg(i_1 = i_2)) \qquad Not(i, o) \triangleq (o = \neg i)$$

- Adding structure:
- Correctness proof: entailment of a specification.

$$(\exists x, Xor(i_1, i_2, x) \wedge Not(x, o)) \implies (o = (i_1 = i_2))$$

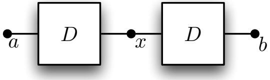
Representing circuits with predicates (or functions).

- Some definitions:

$$Xor(i_1, i_2, o) \triangleq (o = \neg(i_1 = i_2)) \qquad Not(i, o) \triangleq (o = \neg i)$$

- Adding structure:

$$D(a, x) \wedge D(x, b)$$



- Correctness proof: entailment of a specification.

$$(\exists x, Xor(i_1, i_2, x) \wedge Not(x, o)) \implies (o = (i_1 = i_2))$$

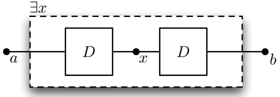
Representing circuits with predicates (or functions).

- Some definitions:

$$Xor(i_1, i_2, o) \triangleq (o = \neg(i_1 = i_2)) \qquad Not(i, o) \triangleq (o = \neg i)$$

- Adding structure:

$$\exists x, D(a, x) \wedge D(x, b)$$



- Correctness proof: entailment of a specification.

$$(\exists x, Xor(i_1, i_2, x) \wedge Not(x, o)) \implies (o = (i_1 = i_2))$$

Representing circuits with predicates (or functions).

- Some definitions:

$$Xor(i_1, i_2, o) \triangleq (o = \neg(i_1 = i_2)) \qquad Not(i, o) \triangleq (o = \neg i)$$

- Adding structure:

Composition

$$D(a, x) \wedge D(x, b)$$

Hiding

$$\exists x, D(a, x) \wedge D(x, b)$$

- Correctness proof: entailment of a specification.

$$(\exists x, Xor(i_1, i_2, x) \wedge Not(x, o)) \implies (o = (i_1 = i_2))$$

Representing circuits with predicates (or functions).

- Some definitions:

$$Xor(i_1, i_2, o) \triangleq (o = \neg(i_1 = i_2)) \qquad Not(i, o) \triangleq (o = \neg i)$$

- Adding structure:

Composition

$$D(a, x) \wedge D(x, b)$$

Hiding

$$\exists x, D(a, x) \wedge D(x, b)$$

- Correctness proof: entailment of a specification.

$$(\exists x, Xor(i_1, i_2, x) \wedge Not(x, o)) \implies (o = (i_1 = i_2))$$

The good points of a shallow embedding

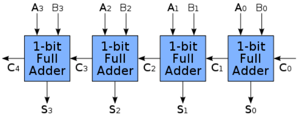
Representing circuits with predicates of the host language makes modelling of circuits easy.

- Use the binders of the theorem prover: ∀, ∃.
- Use function applications to deal with substitution.
- Use recursion to define recursive structure:

```
let rec mux n (sel,a,b,out) = match n with
| 0   → ⊤
| S n → hd out = (if sel then hd a else hd b)
      ∧ mux n (sel,tl a, tl b, tl out)
```

Use lists to model bit-vectors. We have `a`, `b`, `out`: `bool list`.

The bad points of a shallow embedding



Let's define a recursive adder.

- Use recursion to define recursive structure:

```
let rec adder n (a,b,cin,sum,cout) = match n with
| 0   → ⊤
| S n →  ∃ c. adder n (tl a, tl b, c, tl sum, cout)
        ∧ add1 (hd a, hd b, cin, hd sum, c)
```

- Use recursive functions as base blocks:

```
let adder (a,b,cin,sum,cout) =
let cout',sum' = List.fold_right2 (λ a b (c,res) → ... ) a b (cin,[]) in
sum = sum' ∧ cout = cout';;
```

Question

What is a circuit ?

Shallow-Embeddings vs Deep-Embedding

Using a shallow-embedding, there is no way to:

- restrict the quantification on circuits;
- reason on the structure of the circuit in the proof assistant;
- restrict the use of arbitrary functions as basic blocs.

Move to a deep-embedding:

- define a data structure for circuits;
- define what's a circuit semantics (via an interpretation function);
- prove that a device implements a given specification.

Some related work

Ghica, Lafont, ...

Shallow-Embeddings vs Deep-Embedding

Using a shallow-embedding, there is no way to:

- restrict the quantification on circuits;
- reason on the structure of the circuit in the proof assistant;
- restrict the use of arbitrary functions as basic blocs.

Move to a deep-embedding:

- define a data structure for circuits;
- define what's a circuit semantics (via an interpretation function);
- prove that a device implements a given specification.

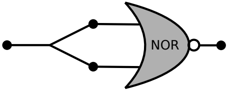
Some related work

Ghica, Lafont, ...

This talk

- Use Coq to embed a language for (synchronous) circuits
- Prove the functionnal correction of circuits

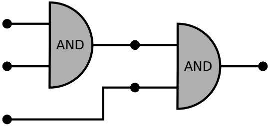
No currents, no delays



Fork 2 ▷ Atom NOR

Ser 1 2 1 (Fork 2) (Atom NOR)

Gate Not : circuit 1 1



(And & One 1) ▷ And

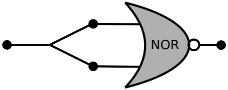
Ser 3 2 1 (Par 2 1 1 1 AND (One 1)) AND

Gate And3 : circuit 3 1

This talk

- Use Coq to embed a language for (synchronous) circuits
- Prove the functionnal correction of circuits

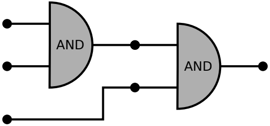
No currents, no delays



Fork 2 ▷ Atom NOR

Ser 1 2 1 (Fork 2) (Atom NOR)

Gate Not : `circuit 1 1`



(And & One 1) ▷ And

Ser 3 2 1 (Par 2 1 1 1 AND (One 1)) AND

Gate And3 : `circuit 3 1`

Outline

1. Defining a deep-embedding of circuits
2. Recursive circuits
3. Sequential circuits: time and loops
4. Corollaries
5. Conclusion, perspectives and related works

A dependent type for circuits in Coq

First version:

- Definition of circuits

  `Inductive` $\mathbb{C}$: `nat` $\rightarrow$ `nat` $\rightarrow$ `Type` := ...

- An n-bit adder as type $\mathbb{C}\ (2 * n + 1)\ (n + 1)$.
- Does not give much structure!

A better dependent type for circuits in Coq

We use arbitrary types as indexes for the ports:

`Inductive` $\mathbb{C}$: `Type` $\rightarrow$ `Type` $\rightarrow$ `Type` := ...

For instance (**1** is the unit type, and $\oplus$ is disjoint-sum):

- `Not` : $\mathbb{C}\ \mathbf{1}\ \mathbf{1}$
- `And3` : $\mathbb{C}\ (\mathbf{1} \oplus \mathbf{1} \oplus \mathbf{1})\ \mathbf{1}$
- `Adder` n : $\mathbb{C}\ (n \cdot \mathbf{1} \oplus n \cdot \mathbf{1} \oplus \mathbf{1})\ (n \cdot \mathbf{1} \oplus \mathbf{1})$

A better dependent type for circuits in Coq

We use arbitrary types as indexes for the ports:

`Inductive` $\mathbb{C}$: `Type` $\rightarrow$ `Type` $\rightarrow$ `Type` := ...

For instance (**1** is the unit type, and $\oplus$ is disjoint-sum):

- `Not` : $\mathbb{C}\ \mathbf{1}\ \mathbf{1}$
- `And3` : $\mathbb{C}\ (\mathbf{1} \oplus \mathbf{1} \oplus \mathbf{1})\ \mathbf{1}$
- `Adder` $n : \mathbb{C}\ (n \cdot \mathbf{1} \oplus n \cdot \mathbf{1} \oplus \mathbf{1})\ (n \cdot \mathbf{1} \oplus \mathbf{1})$

A better dependent type for circuits in Coq

We use arbitrary types as indexes for the ports:

$\texttt{Inductive}\ \mathbb{C}\ : \texttt{Type} \to \texttt{Type} \to \texttt{Type} := ...$

For instance (**1** is the unit type, and $\oplus$ is disjoint-sum):

- $\texttt{Not} : \mathbb{C}\ \mathbf{1}_{i_1}\ \mathbf{1}_{o}$
- $\texttt{And3} : \mathbb{C}\ (\mathbf{1}_{i_1} \oplus \mathbf{1}_{i_2} \oplus \mathbf{1}_{i_3})\ \mathbf{1}_{o}$
- $\texttt{Adder}\ n : \mathbb{C}\ (n \cdot \mathbf{1}_{\mathtt{a}} \oplus n \cdot \mathbf{1}_{\mathtt{b}} \oplus \mathbf{1}_{cin})\ (n \cdot \mathbf{1}_{\mathtt{s}} \oplus \mathbf{1}_{cout})$

Note

The indices are tags, used to identify **1**. (Can use any infinite type.)

A better dependent type for circuits in Coq

We use arbitrary types as indexes for the ports:

`Inductive` $\mathbb{C}$: `Type` → `Type` → `Type` := ...

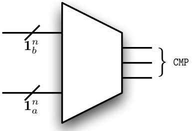
For instance (**1** is the unit type, and ⊕ is disjoint-sum):

- `Not` : $\mathbb{C}\ \mathbf{1}_{i_1}\ \mathbf{1}_o$
- `And3` : $\mathbb{C}\ (\mathbf{1}_{i_1} \oplus \mathbf{1}_{i_2} \oplus \mathbf{1}_{i_3})\ \mathbf{1}_o$
- `Adder` $n : \mathbb{C}\ (n \cdot \mathbf{1}_{\mathtt{a}} \oplus n \cdot \mathbf{1}_{\mathtt{b}} \oplus \mathbf{1}_{cin})\ (n \cdot \mathbf{1}_{\mathtt{s}} \oplus \mathbf{1}_{cout})$

Note

The indices are tags, used to identify **1**. (Can use any infinite type.)

Can use other types. `Compare` $n : \mathbb{C}\ (n \cdot \mathbf{1}_{\mathtt{a}} \oplus n \cdot \mathbf{1}_{\mathtt{b}})$ (`CMP`) where
`Inductive CMP : Type := | Eq | Lt | Gt.`



Plugs

We use circuit combinators ($\&, \triangleright$).

- The information flow is implicit.
- Nameless setting: ports have to be duplicated and reordered using plugs.
- A plug is a circuit of type $\mathbb{C}\ n\ m$ ... defined as a map from m to n.
- Forbids short-circuits.

Example

$\mathbb{C}\ (n \oplus m)\ m$

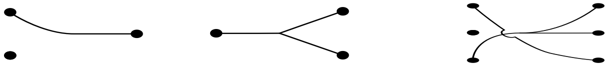
$$\begin{array}{ccc} m & \rightarrow & n \oplus m \\ x & \mapsto & \texttt{inr}\ x \end{array}$$

Plugs

We use circuit combinators ($\&, \triangleright$).

- The information flow is implicit.
- Nameless setting: wires have to be forked, and reordered using plugs.
- A plug is a circuit of type $\mathbb{C}\ n\ m$ ... defined as a map from m to n.
- Forbids short-circuits.
- Examples:

(a) (b) (c)



$\mathbb{C}\ (n \oplus m)\ m$ $\quad$ $\mathbb{C}\ n\ (n \oplus n)$ $\quad$ $\mathbb{C}\ (n \oplus m \oplus p)\ (p \oplus (n \oplus n))$

types must be read bottom-up

a) fun (x : m) ⇒ inr n x

b) fun (x : n ⊕ n) ⇒ match x with inl e ⇒ e | inr e ⇒ e end.

c) fun (x : p ⊕ (n ⊕ n)) ⇒ match x with
| inl ep ⇒ inr (n ⊕ m) ep
| inr (inl en) ⇒ inl p (inl m en)
| inr (inr en) ⇒ inl p (inl m en)

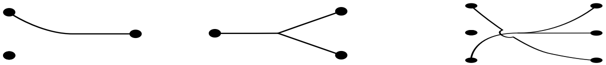
by proof-search

Plugs

We use circuit combinators (&, $\triangleright$).

- The information flow is implicit.
- Nameless setting: wires have to be forked, and reordered using plugs.
- A plug is a circuit of type $\mathbb{C}\ n\ m$ ... defined as a map from m to n.
- Forbids short-circuits.
- Examples:

(a) (b) (c)



$\mathbb{C}\ (n \oplus m)\ m$ $\quad$ $\mathbb{C}\ n\ (n \oplus n)$ $\quad$ $\mathbb{C}\ (n \oplus m \oplus p)\ (p \oplus (n \oplus n))$

types must be read bottom-up

a) fun (x : m) ⇒ inr n x

b) fun (x : n ⊕ n) ⇒ match x with inl e ⇒ e | inr e ⇒ e end.

c) fun (x : p ⊕ (n ⊕ n)) ⇒ match x with
| inl ep ⇒ inr (n ⊕ m) ep
| inr (inl en) ⇒ inl p (inl m en)
| inr (inr en) ⇒ inl p (inl m en)

by proof-search

Abstract syntax

- Strongly typed syntax

```
Inductive ℂ : Type → Type → Type :=
| Atom : ∀ (n m : Type), atom n m → ℂ n m
| Plug : ∀ (n m : Type) (f : m → n), ℂ n m
| Ser  : ∀ (n m p : Type), ℂ n m → ℂ m p → ℂ n p
| Par  : ∀ (n m p q : Type), ℂ n p → ℂ m q → ℂ (n ⊕ m) (p ⊕ q)
| Loop : ∀ (n m p : Type), ℂ (n ⊕ p) (m ⊕ p) → ℂ n m.
```

- Intrinsic approach: an alternative to syntax + typing judgement.

The semantics of a circuit

For a circuit of type $\mathbb{C}\ n\ m$, a relation between $n \rightarrow \mathbb{T}$ and $m \rightarrow \mathbb{T}$.

Rules

$$\textsc{KSer}\ \frac{x \vdash^n_m ins \bowtie middle \qquad y \vdash^m_p middle \bowtie outs}{x \triangleright y \vdash^n_p ins \bowtie outs}$$

$$\textsc{KPar}\ \frac{x \vdash^n_p \texttt{left}\ ins \bowtie \texttt{left}\ outs \qquad y \vdash^m_q \texttt{right}\ ins \bowtie \texttt{right}\ outs}{x \& y \vdash^{n \oplus m}_{p \oplus q} ins \bowtie outs}$$

$$\textsc{KPlug}\ \frac{}{\texttt{Plug}\ f \vdash^n_m ins \bowtie \texttt{lift}\ f\ ins} \qquad \textsc{KLoop}\ \frac{x \vdash^{n \oplus p}_{m \oplus p} \mathrm{app}\ ins\ r \bowtie \mathrm{app}\ outs\ r}{\texttt{Loop}\ x \vdash^n_m ins \bowtie outs}$$

Parametric in the base doors, the type $\mathbb{T}$ and the semantics of the base doors.

Operations

```
Definition left n m : ((n ⊕ m) → 𝕋) → (n → 𝕋) := ...
Definition app n m : (n → 𝕋) → (m → 𝕋) → (n ⊕ m → 𝕋) := ...
Definition lift n m m (f : m → n) : ( n → 𝕋) → (m → 𝕋) := ins ∘ f.
```

The need for abstraction

The semantics of a circuit defines precisely the behavior of a circuit:

- is too precise (may leak some internal details);
- is a relation between two functions $n \to \mathbb{T}$ and $m \to \mathbb{T}$. (Example: $\mathbf{1} \oplus \mathbf{1} \to \mathbb{B}$...).

Use type isomorphisms as "lenses":

```
Class Iso (A B : Type) :={
 iso : A → B;
 uniso : B → A}.
```

```
Class Iso_Props {A B: Type} (I : Iso A B):= {
 iso_uniso : ∀ (x : B), iso (uniso x) = x;
 uniso_iso : ∀ (x : A), uniso (iso x) = x}.
```

Examples:

$$\iota_x \; \frac{}{\mathbf{1}_x \to \mathbb{T} \cong \mathbb{T}} \qquad \cdot \bullet \cdot \; \frac{A \to \mathbb{T} \cong \sigma \qquad B \to \mathbb{T} \cong \tau}{A \oplus B \to \mathbb{T} \cong (\sigma \times \tau)} \qquad \frac{A \to \mathbb{T} \cong \sigma}{n \cdot A \to \mathbb{T} \cong \mathtt{vector}\ \sigma\ \mathtt{n}}$$

The semantics of a circuit defines precisely the behavior of a circuit:

- is too precise (may leak some internal details);
- is a relation between two functions $n \to \mathbb{T}$ and $m \to \mathbb{T}$. (Example: $\mathbf{1} \oplus \mathbf{1} \to \mathbb{B}$...).

Use type isomorphisms as “lenses”:

```
Class Iso (A B : Type) :={
 iso : A → B;
 uniso : B → A}.
```

```
Class Iso_Props {A B: Type} (I : Iso A B):= {
 iso_uniso : ∀ (x : B), iso (uniso x) = x;
 uniso_iso : ∀ (x : A), uniso (iso x) = x}.
```

Examples:

$$\iota_x \ \frac{}{\mathbf{1}_x \to \mathbb{T} \cong \mathbb{T}} \qquad \cdot \bullet \cdot \ \frac{A \to \mathbb{T} \cong \sigma \qquad B \to \mathbb{T} \cong \tau}{A \oplus B \to \mathbb{T} \cong (\sigma \times \tau)} \qquad \frac{A \to \mathbb{T} \cong \sigma}{n \cdot A \to \mathbb{T} \cong \mathtt{vector}\ \sigma\ \mathtt{n}}$$

Putting it all together

We define type classes for abstractions (and modular proofs).

Useful for proof automation

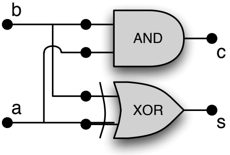
```
Context (n m N M : Type) (Rn : (n→ 𝕋) ≅ N) (Rm : (m→ 𝕋) ≅ M).

Class Realise (c : ℂ n m) (R : N → M → Prop) :=
realise: ∀ ins outs, c ⊢ₘⁿ ins ⋈ outs → R (Rn.iso ins) (Rm.iso outs).

Class Implement (c : ℂ n m) (f : N → M) :=
implement: ∀ ins outs, c ⊢ₘⁿ ins ⋈ outs → Rm.iso outs = f (Rn.iso ins).
```

"Up-to isomorphisms, a given circuit implements a given function."

A complete example

Definition HADD : $\mathbb{C}\ (\mathbf{1}_a \oplus \mathbf{1}_b)\ (\mathbf{1}_s \oplus \mathbf{1}_c)$:=
Fork 2 $(\mathbf{1}_a \oplus \mathbf{1}_b) \triangleright$ (XOR a b s & AND a b c).

Definition hadd := λ (a,b).(a $\otimes$ b, a $\wedge$ b)



Lemma HADD_Spec : Implement
$(\iota_a \bullet \iota_b)$
$(\iota_s \bullet \iota_c)$
HADD hadd.

I : $\mathbf{1}_a \oplus \mathbf{1}_b \to \mathbb{B}$, O : $\mathbf{1}_s \oplus \mathbf{1}_c \to \mathbb{B}$
H : HADD $\vdash^{\mathbf{1}_a \oplus \mathbf{1}_b}_{\mathbf{1}_s \oplus \mathbf{1}_c}$ I $\bowtie$ O
=====================
@iso $(\iota_s \bullet \iota_c)$ O = hadd (@iso $(\iota_a \bullet \iota_b)$ I)

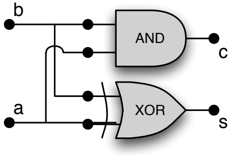
I : $\mathbf{1}_a \oplus \mathbf{1}_b \to \mathbb{B}$, O : $\mathbf{1}_s \oplus \mathbf{1}_c \to \mathbb{B}$
M : $(\mathbf{1}_a \oplus \mathbf{1}_b) \oplus (\mathbf{1}_a \oplus \mathbf{1}_b) \to \mathbb{B}$
H0: iso M = (fun x $\Rightarrow$ (x,x)) (iso I)
H1: iso (left O) = uncurry $\otimes$ (iso (left M))
H2: iso (right O)= uncurry $\wedge$ (iso (right M))
============================
iso O = hadd (iso I)

I: $\mathbb{B} * \mathbb{B}$, O: $\mathbb{B} * \mathbb{B}$,
M : $(\mathbb{B} * \mathbb{B}) * (\mathbb{B} * \mathbb{B})$,
H0: M = (fun x $\Rightarrow$ (x,x)) I
H1: fst O = uncurry $\otimes$ (fst M)
H2: snd O = uncurry $\wedge$ (snd M)
=====================
O = hadd I

A complete example

Definition HADD : $\mathbb{C}$ $(\mathbf{1}_a \oplus \mathbf{1}_b)$ $(\mathbf{1}_s \oplus \mathbf{1}_c)$:=
Fork 2 $(\mathbf{1}_a \oplus \mathbf{1}_b)$ ▷ (XOR a b s & AND a b c).

Definition hadd := λ (a,b).(a ⊗ b, a ∧ b)



Lemma HADD_Spec : Implement
$(\iota_a \bullet \iota_b)$
$(\iota_s \bullet \iota_c)$
HADD hadd.

I : $\mathbf{1}_a \oplus \mathbf{1}_b \rightarrow \mathbb{B}$, O : $\mathbf{1}_s \oplus \mathbf{1}_c \rightarrow \mathbb{B}$
H : HADD $\vdash^{\mathbf{1}_a \oplus \mathbf{1}_b}_{\mathbf{1}_s \oplus \mathbf{1}_c}$ I ⋈ O
=====================
@iso $(\iota_s \bullet \iota_c)$ O = hadd (@iso $(\iota_a \bullet \iota_b)$ I)

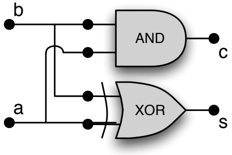
I : $\mathbf{1}_a \oplus \mathbf{1}_b \rightarrow \mathbb{B}$, O : $\mathbf{1}_s \oplus \mathbf{1}_c \rightarrow \mathbb{B}$
M : $(\mathbf{1}_a \oplus \mathbf{1}_b) \oplus (\mathbf{1}_a \oplus \mathbf{1}_b) \rightarrow \mathbb{B}$
H0: iso M = (fun x ⇒ (x,x)) (iso I)
H1: iso (left O) = uncurry ⊗ (iso (left M))
H2: iso (right O)= uncurry ∧ (iso (right M))
============================
iso O = hadd (iso I)

I: $\mathbb{B} * \mathbb{B}$, O: $\mathbb{B} * \mathbb{B}$,
M : $(\mathbb{B} * \mathbb{B}) * (\mathbb{B} * \mathbb{B})$,
H0: M = (fun x ⇒ (x,x)) I
H1: fst O = uncurry ⊗ (fst M)
H2: snd O = uncurry ∧ (snd M)
===================
O = hadd I

A complete example

Definition HADD : $\mathbb{C}$ ($\mathbf{1}_a \oplus \mathbf{1}_b$) ($\mathbf{1}_s \oplus \mathbf{1}_c$) :=
Fork 2 ($\mathbf{1}_a \oplus \mathbf{1}_b$) ▷ (XOR a b s & AND a b c).

Definition hadd := λ (a,b).(a ⊗ b, a ∧ b)



Lemma HADD_Spec : Implement
($\iota_a \bullet \iota_b$)
($\iota_s \bullet \iota_c$)
HADD hadd.

I : $\mathbf{1}_a \oplus \mathbf{1}_b \rightarrow \mathbb{B}$, O : $\mathbf{1}_s \oplus \mathbf{1}_c \rightarrow \mathbb{B}$
H : HADD $\vdash^{\mathbf{1}_a\oplus\mathbf{1}_b}_{\mathbf{1}_s\oplus\mathbf{1}_c}$ I ⋈ O
====================
@iso ($\iota_s \bullet \iota_c$) O = hadd (@iso ($\iota_a \bullet \iota_b$) I)

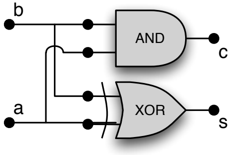
I : $\mathbf{1}_a \oplus \mathbf{1}_b \rightarrow \mathbb{B}$, O : $\mathbf{1}_s \oplus \mathbf{1}_c \rightarrow \mathbb{B}$
M : $(\mathbf{1}_a \oplus \mathbf{1}_b) \oplus (\mathbf{1}_a \oplus \mathbf{1}_b) \rightarrow \mathbb{B}$
H0: iso M = (fun x ⇒ (x,x)) (iso I)
H1: iso (left O) = uncurry ⊗ (iso (left M))
H2: iso (right O)= uncurry ∧ (iso (right M))
==========================
iso O = hadd (iso I)

I: $\mathbb{B} * \mathbb{B}$, O: $\mathbb{B} * \mathbb{B}$,
M : $(\mathbb{B} * \mathbb{B}) * (\mathbb{B} * \mathbb{B})$,
H0: M = (fun x ⇒ (x,x)) I
H1: fst O = uncurry ⊗ (fst M)
H2: snd O = uncurry ∧ (snd M)
==================
O = hadd I

A complete example

Definition HADD : $\mathbb{C}$ $(\mathbf{1}_a \oplus \mathbf{1}_b)$ $(\mathbf{1}_s \oplus \mathbf{1}_c)$:=
Fork 2 $(\mathbf{1}_a \oplus \mathbf{1}_b)$ ▷ (XOR a b s & AND a b c).

Definition hadd := λ (a,b).(a ⊗ b, a ∧ b)



Lemma HADD_Spec : Implement
$(\iota_a \bullet \iota_b)$
$(\iota_s \bullet \iota_c)$
HADD hadd.

I : $\mathbf{1}_a \oplus \mathbf{1}_b \to \mathbb{B}$, O : $\mathbf{1}_s \oplus \mathbf{1}_c \to \mathbb{B}$
H : HADD $\vdash^{\mathbf{1}_a \oplus \mathbf{1}_b}_{\mathbf{1}_s \oplus \mathbf{1}_c}$ I ⋈ O
=====================
@iso $(\iota_s \bullet \iota_c)$ O = hadd (@iso $(\iota_a \bullet \iota_b)$ I)

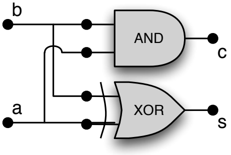
I : $\mathbf{1}_a \oplus \mathbf{1}_b \to \mathbb{B}$, O : $\mathbf{1}_s \oplus \mathbf{1}_c \to \mathbb{B}$
M : $(\mathbf{1}_a \oplus \mathbf{1}_b) \oplus (\mathbf{1}_a \oplus \mathbf{1}_b) \to \mathbb{B}$
H0: iso M = (fun x ⇒ (x,x)) (iso I)
H1: iso (left O) = uncurry ⊗ (iso (left M))
H2: iso (right O)= uncurry ∧ (iso (right M))
===========================
iso O = hadd (iso I)

I: $\mathbb{B} * \mathbb{B}$, O: $\mathbb{B} * \mathbb{B}$,
M : $(\mathbb{B} * \mathbb{B}) * (\mathbb{B} * \mathbb{B})$,
H0: M = (fun x ⇒ (x,x)) I
H1: fst O = uncurry ⊗ (fst M)
H2: snd O = uncurry ∧ (snd M)
===================
O = hadd I

A complete example

Definition HADD : $\mathbb{C}$ $(\mathbf{1}_a \oplus \mathbf{1}_b)$ $(\mathbf{1}_s \oplus \mathbf{1}_c)$:=
Fork 2 $(\mathbf{1}_a \oplus \mathbf{1}_b)$ ▷ (XOR a b s & AND a b c).

Definition hadd := λ (a,b).(a ⊗ b, a ∧ b)



Lemma HADD_Spec : Implement
$(\iota_a \bullet \iota_b)$
$(\iota_s \bullet \iota_c)$
HADD hadd.

I : $\mathbf{1}_a \oplus \mathbf{1}_b \rightarrow \mathbb{B}$, O : $\mathbf{1}_s \oplus \mathbf{1}_c \rightarrow \mathbb{B}$
H : HADD $\vdash^{\mathbf{1}_a \oplus \mathbf{1}_b}_{\mathbf{1}_s \oplus \mathbf{1}_c}$ I ⋈ O
====================
@iso $(\iota_s \bullet \iota_c)$ O = hadd (@iso $(\iota_a \bullet \iota_b)$ I)

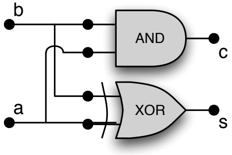
I : $\mathbf{1}_a \oplus \mathbf{1}_b \rightarrow \mathbb{B}$, O : $\mathbf{1}_s \oplus \mathbf{1}_c \rightarrow \mathbb{B}$
M : $(\mathbf{1}_a \oplus \mathbf{1}_b) \oplus (\mathbf{1}_a \oplus \mathbf{1}_b) \rightarrow \mathbb{B}$
H0: iso M = (fun x ⇒ (x,x)) (iso I)
H1: iso (left O) = uncurry ⊗ (iso (left M))
H2: iso (right O)= uncurry ∧ (iso (right M))
==========================
iso O = hadd (iso I)

I: $\mathbb{B} * \mathbb{B}$, O: $\mathbb{B} * \mathbb{B}$,
M : $(\mathbb{B} * \mathbb{B}) * (\mathbb{B} * \mathbb{B})$,
H0: M = (fun x ⇒ (x,x)) I
H1: fst O = uncurry ⊗ (fst M)
H2: snd O = uncurry ∧ (snd M)
==================
O = hadd I

A complete example

Definition HADD : $\mathbb{C}$ $(\mathbf{1}_a \oplus \mathbf{1}_b)$ $(\mathbf{1}_s \oplus \mathbf{1}_c)$:=
Fork 2 $(\mathbf{1}_a \oplus \mathbf{1}_b)$ ▷ (XOR a b s & AND a b c).

Definition hadd := λ (a,b).(a ⊗ b, a ∧ b)



Lemma HADD_Spec : Implement
$(\iota_a \bullet \iota_b)$
$(\iota_s \bullet \iota_c)$
HADD hadd.

I : $\mathbf{1}_a \oplus \mathbf{1}_b \to \mathbb{B}$, O : $\mathbf{1}_s \oplus \mathbf{1}_c \to \mathbb{B}$
H : HADD $\vdash^{\mathbf{1}_a \oplus \mathbf{1}_b}_{\mathbf{1}_s \oplus \mathbf{1}_c}$ I ⋈ O
====================
@iso $(\iota_s \bullet \iota_c)$ O = hadd (@iso $(\iota_a \bullet \iota_b)$ I)

I : $\mathbf{1}_a \oplus \mathbf{1}_b \to \mathbb{B}$, O : $\mathbf{1}_s \oplus \mathbf{1}_c \to \mathbb{B}$
M : $(\mathbf{1}_a \oplus \mathbf{1}_b) \oplus (\mathbf{1}_a \oplus \mathbf{1}_b) \to \mathbb{B}$
H0: iso M = (fun x ⇒ (x,x)) (iso I)
H1: iso (left O) = uncurry ⊗ (iso (left M))
H2: iso (right O)= uncurry ∧ (iso (right M))
============================
iso O = hadd (iso I)

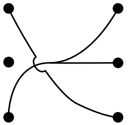
I: $\mathbb{B} * \mathbb{B}$, O: $\mathbb{B} * \mathbb{B}$,
M : $(\mathbb{B} * \mathbb{B}) * (\mathbb{B} * \mathbb{B})$,
H0: M = (fun x ⇒ (x,x)) I
H1: fst O = uncurry ⊗ (fst M)
H2: snd O = uncurry ∧ (snd M)
==================
O = hadd I

One more word on Plugs

- Thanks to the use of tags, plugs can be defined by proof search.
- ... but, for each plug, we have to exhibit the fonction it implements (up to isos).
- A better solution for simple plugs is to use the following definition.

```
Inductive monoid : Type :=
| Var : Type → monoid
| • : monoid → monoid → monoid.
```

```
Inductive ⊢ : monoid → monoid → Set :=
| M : ∀ A B C, A ⊢ C → B ⊢ C → (A • B) ⊢ C
| L : ∀ A B C, A ⊢ B → A ⊢ (B • C)
| R : ∀ A B C, A ⊢ B → A ⊢ (C • B)
| I : ∀ A, A ⊢ A.
```



$\mathbb{C}\ (n \oplus m \oplus p)\ (p \oplus (n \oplus n))$

$$\dfrac{\dfrac{I}{p \vdash (n \bullet m) \bullet p}\,R \qquad \dfrac{\dfrac{\dfrac{I \quad I}{n \bullet n \vdash n}\,M}{n \bullet n \vdash n \bullet m}\,L}{n \bullet n \vdash (n \bullet m) \bullet p}\,L}{p \bullet (n \bullet n) \vdash (n \bullet m) \bullet p}\,M$$

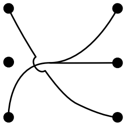
- Evaluated to $(p \oplus (n \oplus n)) \to (n \oplus m \oplus p)$ (the plug)
- Evaluated to $(\overline{n} \otimes \overline{m} \otimes \overline{p}) \to (\overline{p} \otimes (\overline{n} \otimes \overline{n}))$ (the action of the plug on values)

One more word on Plugs

- Thanks to the use of tags, plugs can be defined by proof search.
- ... but, for each plug, we have to exhibit the fonction it implements (up to isos).
- A better solution for simple plugs is to use the following definition.

```
Inductive monoid : Type :=
| Var : Type → monoid
| • : monoid → monoid → monoid.
```

```
Inductive ⊢ : monoid → monoid → Set :=
| M : ∀ A B C, A ⊢ C → B ⊢ C → (A • B) ⊢ C
| L : ∀ A B C, A ⊢ B → A ⊢ (B • C)
| R : ∀ A B C, A ⊢ B → A ⊢ (C • B)
| I : ∀ A, A ⊢ A.
```



$\mathbb{C}\ (n \oplus m \oplus p)\ (p \oplus (n \oplus n))$

$$\text{M}\ \dfrac{\text{R}\ \dfrac{I}{p \vdash (n \bullet m) \bullet p} \qquad \text{L}\ \dfrac{\text{L}\ \dfrac{\text{M}\ \dfrac{I \qquad I}{n \bullet n \vdash n}}{n \bullet n \vdash n \bullet m}}{n \bullet n \vdash (n \bullet m) \bullet p}}{p \bullet (n \bullet n) \vdash (n \bullet m) \bullet p}$$

- Evaluated to $(p \oplus (n \oplus n)) \rightarrow (n \oplus m \oplus p)$ (the plug)
- Evaluated to $(\overline{n} \otimes \overline{m} \otimes \overline{p}) \rightarrow (\overline{p} \otimes (\overline{n} \otimes \overline{n}))$ (the action of the plug on values)

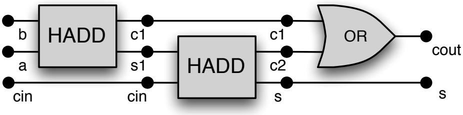
Outline

1. Defining a deep-embedding of circuits
2. Recursive circuits
3. Sequential circuits: time and loops
4. Corollaries
5. Conclusion, perspectives and related works

n-bit integers

```
Record W_n := mk_word {val : Z; range: 0 ≤ val < 2^n}.

Definition repr n : Z → W_n := ...
Definition high n m : W_(n+m) → W_m := ...
Definition low n m : W_(n+m) → W_n := ...
Definition combine n m : W_n → W_m → W_(n+m) := ...
Definition carry_add n (x y : W_n) (b: B) : W_n * B :=
    let e := val x + val y + (if b then 1 else 0) in (e mod 2^n, 2^n ≤ e)

Definition Φ^n_x : (n · 1_x → B) ≅ (W_n) := ...
```

A 1-bit adder

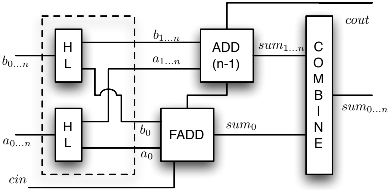


Context a b cin sum cout : string.
Program Definition FADD :
$\mathbb{C}$ ($\mathbf{1}_{cin} \oplus (\mathbf{1}_a \oplus \mathbf{1}_b)$) ($\mathbf{1}_{sum} \oplus \mathbf{1}_{cout}$):=
(ONE $\mathbf{1}_{cin}$ & HADD a b "s1" "c1")
▷ ...
▷ (HADD cin "s1" sum "c2" & ONE $\mathbf{1}_{"c1"}$)
▷ ...
▷ (ONE $\mathbf{1}_{sum}$ & OR "c2" "c1" cout).

Instance FADD_1 : Implement
$(\iota_{cin} \bullet (\iota_a \bullet \iota_b))$
$(\iota_{sum} \bullet \iota_{cout})$
FADD
(fun (c,(x,y)) ⇒ (x ⊕ (y ⊕ c),(x ∧ y) ∨ c ∧ (x ⊕ y))).

Instance FADD_2 : Implement
$(\iota_{cin} \bullet (\Phi^1_a \bullet \Phi^1_b))$
$(\Phi^1_{sum} \bullet \iota_{cout})$
FADD
(fun (c,(x,y)) ⇒ carry_add 1 x y c).

A n-bit adder

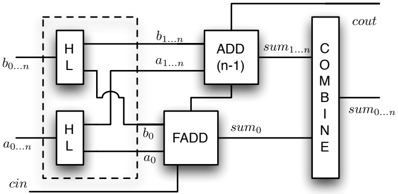


Program Fixpoint ADD cin a b cout sum n :
$\mathbb{C}\ (\mathbf{1}_{\texttt{cin}} \oplus n \cdot \mathbf{1}_{\texttt{a}} \oplus n \cdot \mathbf{1}_{\texttt{b}})\ (n \cdot \mathbf{1}_{\texttt{sum}} \oplus \mathbf{1}_{\texttt{cout}}) :=$
match n with
  | 0 ⇒ ...
  | S p ⇒ ... ▷ (ONE ($\mathbf{1}_{\texttt{cin}}$) & HIGHLOWS a b 1 p)
  ▷ ... ▷ (FADD a b cin sum "c" & ONE ($p \cdot \mathbf{1}_{\texttt{a}} \oplus p \cdot \mathbf{1}_{\texttt{b}}$))
  ▷ ... ▷ (ONE ($\mathbf{1}_{sum}$) & ADD "c" a b cout s p)
  ▷ ... ▷ COMBINE sum 1 p & ONE ($\mathbf{1}_{\texttt{cout}}$)
end.

Lemma add_parts n m (xH yH: $\mathbb{W}_{\texttt{m}}$) (xL yL : $\mathbb{W}_{\texttt{n}}$) cin:
let (sumL,middle) := carry_add n xL yL cin in
let (sumH,cout) := carry_add m xH yH middle in
let sum := combine n m sumL sumH in
 carry_add (n + m) (combine n m xL xH)(combine n m yL yH) cin = (sum,cout).

Instance ADD_Spec cin a b cout sum n : Implement
$(\iota_{\texttt{cin}} \bullet (\Phi^{\texttt{n}}_{\texttt{a}} \bullet \Phi^{\texttt{n}}_{\texttt{b}}))$
$(\Phi^{\texttt{n}}_{\texttt{sum}} \bullet \iota_{\texttt{cout}})$
(ADD cin a b cout sum n)
(fun (c,(x,y)) ⇒ carry_add c x y).

Some sub-components



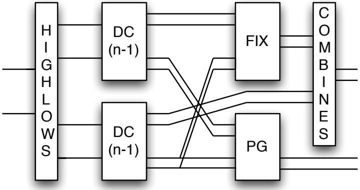
Definition HL x n p : $\mathbb{C}\ ((n+p)\cdot \mathbf{1}_x)\ (n\cdot \mathbf{1}_x \oplus p\cdot \mathbf{1}_x)$:= Plug ...
Definition COMBINE x n p : $\mathbb{C}\ (n\cdot \mathbf{1}_x \oplus p\cdot \mathbf{1}_x)\ ((n+p)\cdot \mathbf{1}_x)$:= Plug ...

Instance HL_Spec x n p: Implement
$(\Phi_x^{n+p})\ (\Phi_x^n \bullet \Phi_x^p)$ (HL x n p) (fun x ⇒ (low n p x, high n p x)).

Instance COMBINE_Spec x n p: Implement
$(\Phi_x^n \bullet \Phi_x^p)\ (\Phi_x^{n+p})$ (COMBINE x n p) (fun x ⇒ (combine n p (fst x) (snd x))).

A divide and conquer adder (1)

- Add in parallel the high-order and the low-order bits.
- Computes s (resp. t) the sum without (resp. with) a carry-in
- Computes p the carry-propagate and g the carry-generate

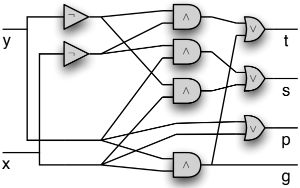


Implements the following Coq (high-level) function:

Definition dc n : $\mathbb{W}_{2^n} * \mathbb{W}_{2^n} \rightarrow \mathbb{B} * \mathbb{B} * \mathbb{W}_{2^n} * \mathbb{W}_{2^n}$:= fun (x,y) ⇒
let (s,g) := carry_add 2^n x y false in
let (t,p) := carry_add 2^n x y true in (g,p,s,t).

A divide and conquer adder (2)

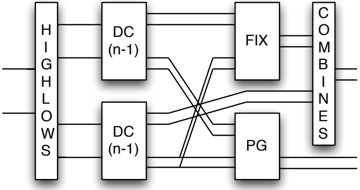
The base case



Implements the following Coq (high-level) function (for $n = 0$)

Definition dc n : $\mathbb{W}_{2^n} * \mathbb{W}_{2^n} \rightarrow \mathbb{B} * \mathbb{B} * \mathbb{W}_{2^n} * \mathbb{W}_{2^n}$:= fun (x,y) ⇒
let (s,g) := carry_add 2^n x y false in
let (t,p) := carry_add 2^n x y true in (g,p,s,t).

A divide and conquer adder (3)



Implements the following Coq (high-level) function:

```
Definition dc n : W_{2^n} * W_{2^n} → B * B * W_{2^n} * W_{2^n} := fun (x,y) ⇒
let (s,g) := carry_add 2^n x y false in
let (t,p) := carry_add 2^n x y true in (g,p,s,t).
```

- The high-level specification says nothing of the computationnal behavior of the circuit.
- The deep-embedding makes it possible to study the latency of this circuit.

Outline

In this section $\mathbb{T}$ is `nat → bool`.
We have several interesting isomorphisms:

```
Definition Iso_stream A B C (I: (A → B) ≅ C) : (A → stream B) ≅ (stream C) := ...
Definition Iso_prod_stream : (stream A * stream B) ≅ (stream (A * B)) := ...
Definition Iso_vector_stream n : (vector (stream A) n) ≅ (stream (vector A n)) := ...
```

Examples:

$(\mathbf{1} \oplus \mathbf{1} \rightarrow \texttt{nat} \rightarrow \mathbb{B}) \cong (\texttt{stream}\ (\mathbb{B} * \mathbb{B}))$
$(n \cdot \mathbf{1} \rightarrow \texttt{nat} \rightarrow \mathbb{B}) \cong (\texttt{stream}\ (\mathbb{W}_{\texttt{n}}))$

We assume (through appropriate parametrization) a gate DFF that implements a `pre`:

```
Definition pre {A} (d : A):
stream A → stream A := fun f t ⇒
match t with | 0 ⇒ d | S p ⇒ f p end.
```

```
Instance DFF_Realise_stream {a out}:
Implement (DFF a out) (ι_a) (ι_out)
(pre false).
```

Streams

In this section $\mathbb{T}$ is `nat → bool`.
We have several interesting isomorphisms:

```
Definition Iso_stream A B C (I: (A → B) ≅ C) : (A → stream B) ≅ (stream C) := ...
Definition Iso_prod_stream : (stream A * stream B) ≅ (stream (A * B)) := ...
Definition Iso_vector_stream n : (vector (stream A) n) ≅ (stream (vector A n)) := ...
```

Examples:

$(\mathbf{1} \oplus \mathbf{1} \to \mathtt{nat} \to \mathbb{B}) \cong (\mathtt{stream}\ (\mathbb{B} * \mathbb{B}))$
$(n \cdot \mathbf{1} \to \mathtt{nat} \to \mathbb{B}) \cong (\mathtt{stream}\ (\mathbb{W}_{\mathtt{n}}))$

We assume (through appropriate parametrization) a gate `DFF` that implements a `pre`:

```
Definition pre {A} (d : A):
stream A → stream A := fun f t ⇒
match t with | 0 ⇒ d | S p ⇒ f p end.
```

```
Instance DFF_Realise_stream {a out}:
Implement (DFF a out) (ι_a) (ι_out)
(pre false).
```

A buffer

```
Variable CELL : ℂ n n.
Fixpoint COMPOSEN k : ℂ n n :=
match k with
| 0 ⇒ Plug id
| S p ⇒ CELL ▷ (COMPOSEN p)
end.
```

```
Variable CELL : ℂ n m.
Fixpoint MAP k : ℂ (n · k) (m · k):=
match k with
| 0 ⇒ Plug id
| S p ⇒ CELL & (MAP p)
end.
```

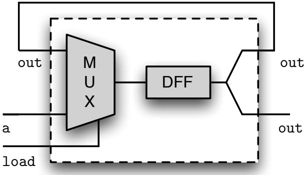
Then, we can define a buffer with parametric width and length:

```
Remark useful_iso : n · 1 → stream 𝔹 ≅ stream (vector 𝔹 n) := ...

Definition FIFO x n k : ℂ (k · 1_x) (k · 1_x) := COMPOSEN (MAP (DFF x x) k) n.
Definition fifo n k (v : stream (vector 𝔹 k)) : stream (vector 𝔹 k) :=
  fun t ⇒ if n < t then v (t − n) else Vector.repeat k false.
```

A memory element

We can also deal with state-holding structures:

$\texttt{Definition REGISTER}: \mathbb{C}\ (\mathbf{1}_{\texttt{load}} \oplus \mathbf{1}_{\texttt{a}})\ \mathbf{1}_{\texttt{out}} :=$
$\texttt{Loop}\ (\mathbf{1}_{\texttt{load}} \oplus \mathbf{1}_{\texttt{a}})\ \mathbf{1}_{\texttt{out}}\ \mathbf{1}_{\texttt{out}}$
$\quad (\texttt{Plug} \ldots \triangleright \texttt{MUX2 a out load "in\_dff"}$
$\qquad \triangleright \texttt{DFF "in\_dff" out}$
$\qquad \triangleright \texttt{Fork 2}\ \mathbf{1}_{\texttt{out}}).$



This circuit realise the following relation:

$\texttt{Instance Register\_Spec} : \texttt{Realise}(\ldots :\ \mathbf{1}_{\texttt{load}} \oplus \mathbf{1}_{\texttt{a}} \to \texttt{stream}\ \mathbb{B} \cong \texttt{stream}\ \mathbb{B} * \mathbb{B})\ (\iota_{\texttt{out}})\ \texttt{REGISTER}$
$(\texttt{fun}\ (\texttt{ins} : \texttt{stream}\ (\mathbb{B} * \mathbb{B}))\ (\texttt{outs} : \texttt{stream}\ \mathbb{B}) \Rightarrow$
$\quad \texttt{outs} = \texttt{pre false}\ (\texttt{fun t} \Rightarrow \texttt{if fst (ins t) then snd (ins t) else outs t})).$

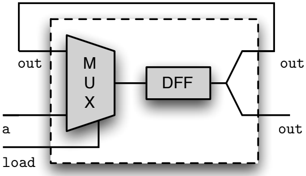
Note

Relations on streams are not the nicest way to reason about state-holding devices.

A memory element

We can also deal with state-holding structures:

Definition REGISTER: $\mathbb{C}$ ($\mathbf{1}_{load} \oplus \mathbf{1}_{a}$) $\mathbf{1}_{out}$:=
Loop ($\mathbf{1}_{load} \oplus \mathbf{1}_{a}$) $\mathbf{1}_{out}$ $\mathbf{1}_{out}$
  (Plug ... ▷ MUX2 a out load "in_dff"
  ▷ DFF "in_dff" out
  ▷ Fork 2 $\mathbf{1}_{out}$).



This circuit realise the following relation:

Instance Register_Spec : Realise(... : $\mathbf{1}_{load} \oplus \mathbf{1}_{a} \rightarrow$ stream $\mathbb{B} \cong$ stream $\mathbb{B} * \mathbb{B}$) (ι_{out}) REGISTER
(fun (ins : stream ($\mathbb{B} * \mathbb{B}$)) (outs : stream $\mathbb{B}$) $\Rightarrow$
  outs = pre false (fun t $\Rightarrow$ if fst (ins t) then snd (ins t) else outs t)).

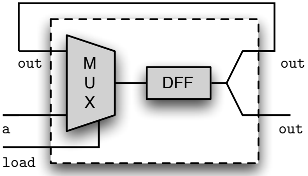
Note

Relations on streams are not the nicest way to reason about state-holding devices.

A memory element

We can also deal with state-holding structures:

Definition REGISTER: $\mathbb{C}$ ($\mathbf{1}_{\text{load}} \oplus \mathbf{1}_{\text{a}}$) $\mathbf{1}_{\text{out}}$:=
Loop ($\mathbf{1}_{\text{load}} \oplus \mathbf{1}_{\text{a}}$) $\mathbf{1}_{\text{out}}$ $\mathbf{1}_{\text{out}}$
  (Plug ... ▷ MUX2 a out load "in_dff"
            ▷ DFF "in_dff" out
            ▷ Fork 2 $\mathbf{1}_{\text{out}}$).



This circuit realise the following relation:

Instance Register_Spec : Realise(... : $\mathbf{1}_{\text{load}} \oplus \mathbf{1}_{\text{a}}$ → stream $\mathbb{B}$ ≅ stream $\mathbb{B} * \mathbb{B}$) (ι_{out}) REGISTER
(fun (ins : stream ($\mathbb{B} * \mathbb{B}$)) (outs : stream $\mathbb{B}$) ⇒
  outs = pre false (fun t ⇒ if fst (ins t) then snd (ins t) else outs t)).

Note

Relations on streams are not the nicest way to reason about state-holding devices.

Lifting combinatorial circuits

Well-behaved circuits

"Circuits that have the same behavior in the boolean setting and pointwise in a stream"

A well-behaved atom satisfies:

$$\forall ins, \forall outs, \mathbb{B} \models ins \bowtie outs \implies \forall t, (\texttt{stream}\ \mathbb{B}) \models ins@t \bowtie outs@t.$$

Plugs are well-behaved, as well as parallel and serial circuits when their sub-circuits are well-behaved.

Lifting

```
Lemma lifting_map n m x (Hwf: wb n m x) N M
(Rn : (n → Data) ≅ N)
(Rm : (m → Data) ≅ M)
(f : N → M):
Implement Data tech_spec Rn Rm x f →
Implement (stream Data) tech_spec' (Iso_stream Rn) (Iso_stream Rm) x (Stream.map f).
```

Simulation and other stuff

Simulation:

- This first-order encoding makes it possible to simulate circuits inside Coq.
- It requires a computational interpretation of each basic gate.
- Not very efficient...
- ```
  Definition test_dc n a b :=
   let init := uniso ... (a, b) in
   match SIM.sim (DC n) with
    | None ⇒ None
    | Some x ⇒ Some (iso .... x)
   end.
  Eval compute in test_dc 2 (Word.repr 4 3) (Word.repr 4 3).
  ```

Using the same idea:

- computation of the gate-count and length of the critical path;
- pretty-printing of the list of gates (and their connection).

Almost to VHDL
  ```

Simulation and other stuff

Simulation:

- This first-order encoding makes it possible to simulate circuits inside Coq.
- It requires a computational interpretation of each basic gate.
- Not very efficient...
- ```
  Definition test_dc n a b :=
   let init := uniso ... (a, b) in
   match SIM.sim (DC n) with
    | None ⇒ None
    | Some x ⇒ Some (iso .... x)
   end.
  Eval compute in test_dc 2 (Word.repr 4 3) (Word.repr 4 3).
  ```

Using the same idea:

- computation of the gate-count and length of the critical path;
- pretty-printing of the list of gates (and their connection).

Almost to VHDL
  ```

Conclusion

- A deep-embedding of circuits in Coq
- Build and reason about circuits, proving high-level specifications through type-isomorphisms
- Dependent types are useful to capture some well-formedness properties
- Examples: arithmetic circuits of parametric size, proved by induction

Related works

- Verifying circuits with theorem provers:
  - in HOL or in Coq, mainly shallow-embeddings;
  - in HOL, a compiler from HOL to VHDL;
  - in ACL2, shallow-embeddings (up to a microprocessor), not higher-order.
- Algebraic definitions of circuits:
  - Lafont: algebraic theory of boolean circuits.
  - Hinze: the algebra of parallel prefix circuits.
- Functional languages in hardware design:
  - Many approaches based on circuits combinators (often lack dependent types)
  - Lava: language embedded in Haskell to describe circuits (combinator based, but use names for wires)

Perspectives

- More arithmetic circuits.
- Use mealy automata rather that stream equations to specify state-holding circuits.
- Some front-end to generate circuits.

When I am PhDone:

```
Inductive aexp : nat → Type :=
| AVar: ∀ n (i : Var E n), aexp n
| AConst: ∀ n, Word.word n → aexp n
| APlus: ∀ n, aexp n → aexp n → aexp n
| ALo: ∀ n m, aexp (n + m) → aexp n
| AHi: ∀ n m, aexp (n + m) → aexp m
| ACat: ∀ n m, aexp n → aexp m → aexp (n + m).
```

```
Inductive bexp : Type :=
| BTrue: bexp
| BFalse: bexp
| BEq: ∀ n, aexp n → aexp n → bexp
| BLt: ∀ n, aexp n → aexp n → bexp
| BNot: bexp → bexp
| BAnd: bexp → bexp → bexp.
```

```
Inductive com (E: list nat) : Type :=
| CSkip: com E
| CAss: ∀ n (v : Var E n) , aexp E n → com E
| CSeq: com E → com E → com E
| CIf: bexp E → com E → com E → com E
| CWhile: bexp E → com E → com E
| CNew: ∀ n, aexp E n → com (snoc E n) → com E
```

Perspectives

- More arithmetic circuits.
- Use mealy automata rather that stream equations to specify state-holding circuits.
- Some front-end to generate circuits.

When I am PhDone:

```
Inductive aexp : nat → Type :=
| AVar: ∀ n (i : Var E n), aexp n
| AConst: ∀ n, Word.word n → aexp n
| APlus: ∀ n, aexp n → aexp n → aexp n
| ALo: ∀ n m, aexp (n + m) → aexp n
| AHi: ∀ n m, aexp (n + m) → aexp m
| ACat: ∀ n m, aexp n → aexp m → aexp (n + m).
```

```
Inductive bexp : Type :=
| BTrue: bexp
| BFalse: bexp
| BEq: ∀ n, aexp n → aexp n → bexp
| BLt: ∀ n, aexp n → aexp n → bexp
| BNot: bexp → bexp
| BAnd: bexp → bexp → bexp.
```

```
Inductive com (E: list nat) : Type :=
| CSkip: com E
| CAss: ∀ n (v : Var E n) , aexp E n → com E
| CSeq: com E → com E → com E
| CIf: bexp E → com E → com E → com E
| CWhile: bexp E → com E → com E
| CNew: ∀ n, aexp E n → com (snoc E n) → com E
```