A Taste of Hardware Verification

From combinational circuits to synchronous circuits, a tour of simple theory, general culture and practical tools

Formal Verification, Fall 2023 - Thomas, EPFL

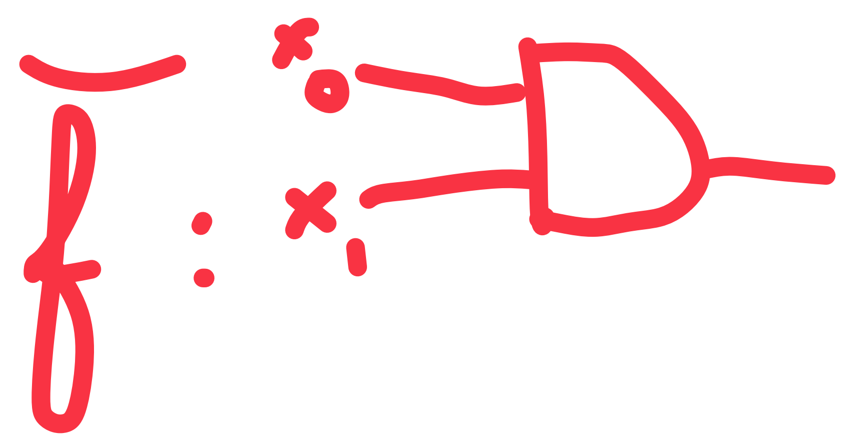
Outline

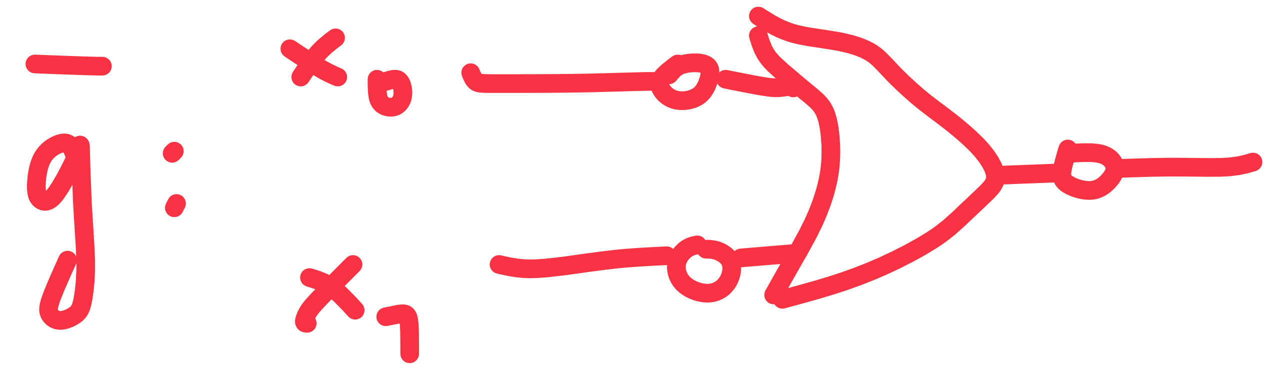
1. Boolean functions, combinational circuits, equivalence
2. Sequential machines and equivalence
3. A better abstraction: more modular way to describe sequential machines
4. Along the way: detour about k-induction, finding and describing simulation relations, maybe inductive propositions

Boolean function, combinational circuit

Reminders of the stateless world

$f, g : \mathbb{B}^n \to \mathbb{B}^m$





$$f \sim g := \forall x \in \mathbb{B}^n, f(x) = g(x)$$

SAT solvers are good at this problem! SAT(!(f x <=> g x))

History: before SAT solvers became efficient (~2000s), people had invented BDDs, a cool DAG datastructure that ensures:

f=g <=> Rep(f) = Ref(g)

Curious minds: 2008 Knuth Christmas Lecture (YouTube)

What is this good for?

$\text{add} : \mathbb{B}^{2n} \to \mathbb{B}^{2n}, (x, y) \mapsto x + y$

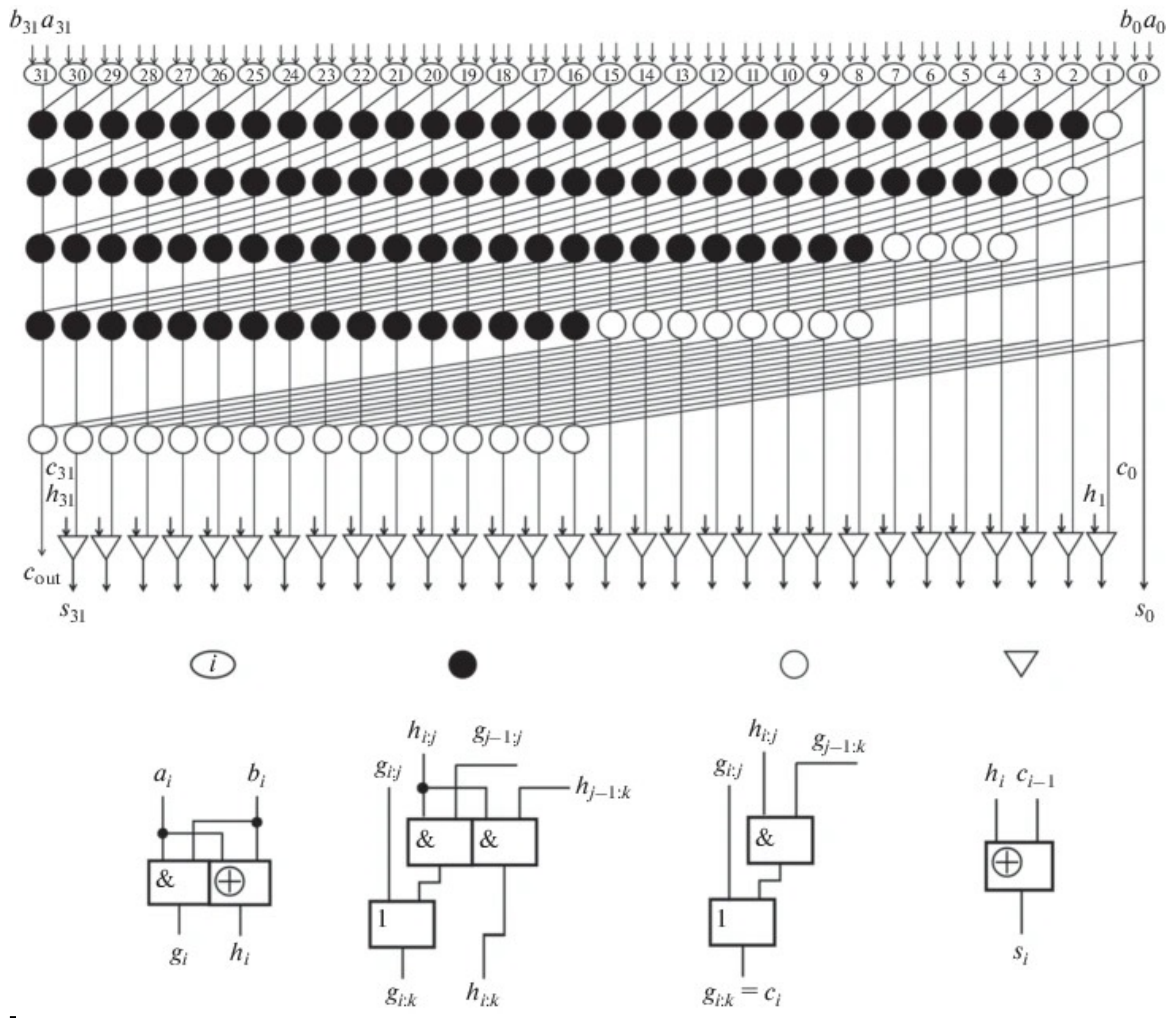
Target quite achievable:

Correctness of the implementation

Observation:

spec can often not directly written as a circuit

spec/impl also can be written targeting a different library



Source picture: *Improving the Operating Efficiency of a Multibit Binary Parallel-Prefix Adder*, Yakunin & al. Russian Microelectronic 2021

Switch to verification playground: prove the correctness of a Barrel shifter

What is a barrel shifter?

Takeaways for combinational ckts

Easy to formalize

Off-the-shelf tools to solve the verification conditions

Very <u>useful</u> for:

- Automatic translation validation in synthesis/flows
- Verify the implementation of nontrivial circuit “algorithms”

Also:

- Super easy, “push-button” to use!

What is this not so good for

Practical issues, leading to interesting theory

Consider a RISC-V processor, internally the ISA specify encoding of instructions:

$$\text{decoder} : \text{inst} \in \mathbb{B}^{32} \mapsto (\text{ins\_type} \in \mathbb{B}^{3}, \text{rs1} \in \mathbb{B}^{5}, \text{rs2} \in \mathbb{B}^{5}, \text{rd} \in \mathbb{B}^{5}, \text{immediate} \in \mathbb{B}^{32})$$

Some ins_type: fields are useless, i.e., jump does not always have an rd

***Boolean** specifications of a decoder will necessarily over specify, we need a weaker form of equivalence!*

Boolean functions are not flexible enough, let's consider $\bar{\mathbb{B}} = \{0,1,\mathrm{X}\}$ equipped with:

$$x \sim_{\bar{\mathbb{B}}} y := (x = y \text{ if } x\ y \in \mathbb{B} \text{ else True})$$

$$f \sim g := \forall x \in \bar{\mathbb{B}}^{n}, f(x) \sim_{\bar{\mathbb{B}}} g(x)$$

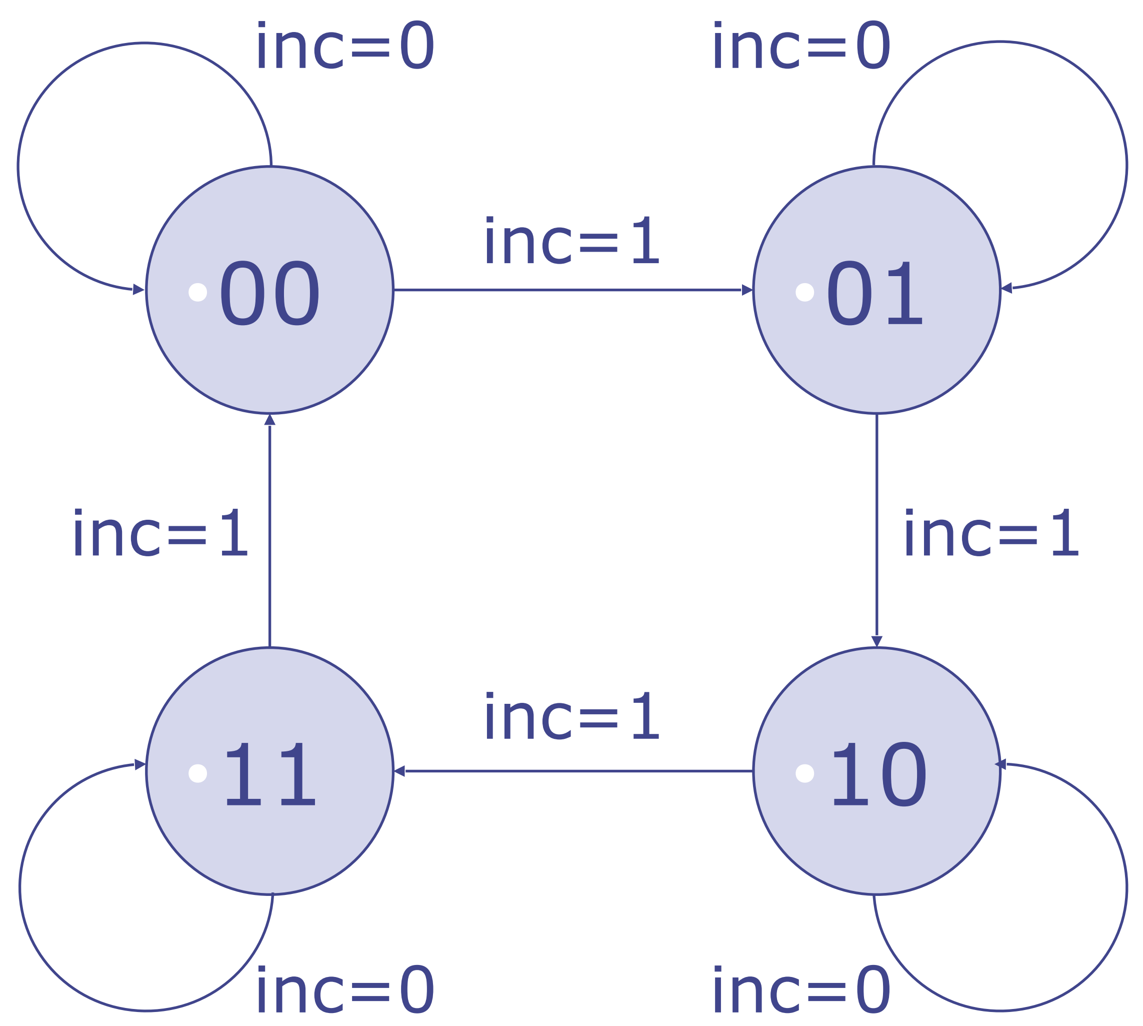
Note: Tri-state logic (0,1,Z) is NOT the same thing. Z is a well-defined value, which means "actively no value"

Sequential Machines

Sequential (Moore) machines/synchronous circuits

Example

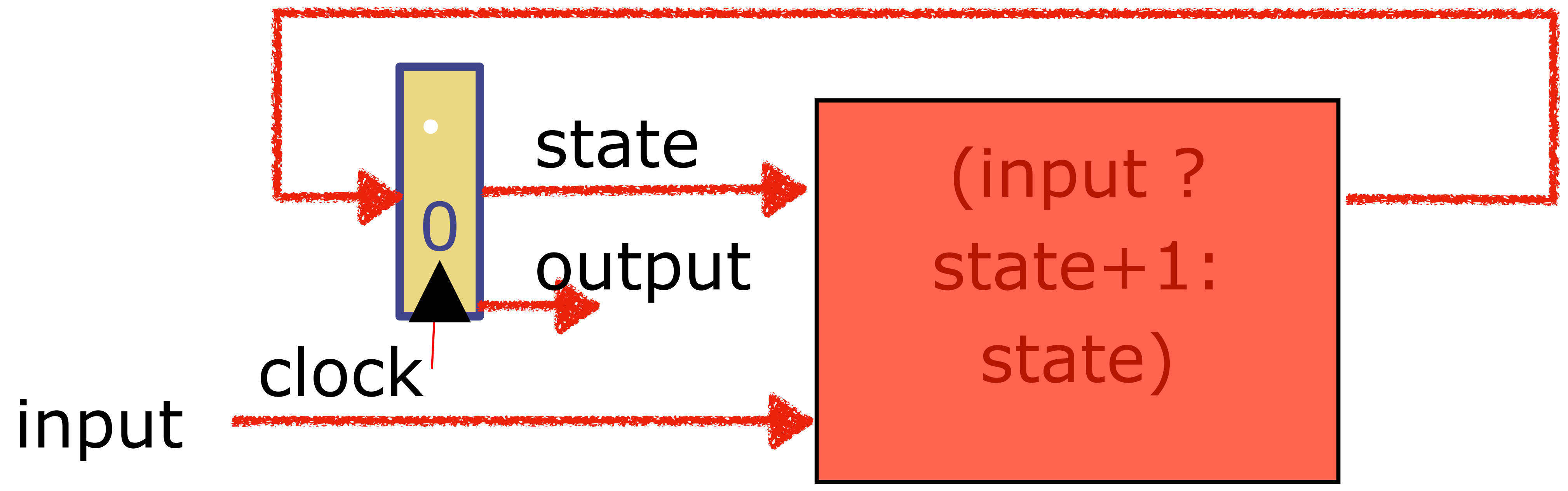
Moore's machine:



Prev State s	NextState s′	
	inc = 0	inc = 1
00	00	01
01	01	10
10	10	11
11	11	00

One realization using a *synchronous circuit*:

```
out reg[1:0] s=0;
if (inc^t == 0)
   q1q0^(t+1) = q1q0^t
 else
   q1q0^(t+1) = q1q0^t + 1;
```



Sequential (Moore) machines

Binary formalization

Moore's machine (*Binarized*) is a collection:

A number of bits for the internal state: N

A number of bits for the input each *step: I*

A number of bits for the output each *step:* O

A reset predicate: Reset : $\mathbb{B}^N \to \mathbb{B}$

A Boolean state update function $u : \mathbb{B}^{N+I} \to \mathbb{B}^N$

An Boolean observation function $o : \mathbb{B}^N \to \mathbb{B}^O$

Proving properties of sequential machines

Grandma's recipe to prove that a property P holds for a synchronous Ckt:

1. Find (N, I, O, r, u, o) corresponding to your synchronous ckt *(trivial)*
2. Find a property Inv such that $\forall x, Inv(x) \Rightarrow P(x)$
3. Prove that $\forall s, \text{Reset}(s) \Rightarrow Inv(s)$ *(not too hard)*
4. Prove that $\forall s\ i, Inv(s) \Rightarrow Inv(u(s, i))$

Even when you care only about a property of the outputs you typically need an inductive invariant about the state.

Automatic invariant discovery!

K-induction

On natural numbers:

Basic induction: forall P, P 0 -> (for all s, P s -> P (s + 1)) -> for all n . P n

2-induction: forall P, P 0 -> P 1 -> (for all s, $\underbrace{\text{P s -> P (s + 1)}}_{\uparrow \ \text{More assumptions available}}$ -> P (s + 2)) -> for all n. P n

Source: *Checking safety properties using induction and a SAT-solver*, Sheeran & al. FMCAD 2000

What is it good for?

Demo number 2: Multiplication by adding, go to other machine (if t > 25mn, I should start panicking)

Practical wisdoms from the hardware and distributed system crowds:

Bugs exist *in the small* reasonable when the code is properly parametrised

Beware of hardcoded value

There are various “engine”, k-induction, IC3, parametrised with various abstractions heuristics, but also

About the example

Shadow logics - Device Under Test

Notice that there is a fine line between the DUT and the verification test bench

The shadow logic should be only shadow and should not influence the design

What we prove is only what the shadow logic said

- What if we have a bug in the shadow logic?
- Shadow logic and design are intertwined
  - One need to be expert both in formal method and in the specifics of the design

How did k-induction work?

The little magic that is easy to miss

In our specific case what is P and what is Inv?

Intuitively, why adding older P work?

Is it different from reaching all state because state space is finite?

Limitation: commonly circuits have large “depth/history”, k-induction is insufficient

Detour to not go to hardware verification jail - another useful formalism

Alternative specification approach - zoo of temporal logics

LTL,CTL, CTL*, TLA, ITL, MTL, PSL, TPTL, HyperLTL ….

A common alternative is to specify temporal logic properties:

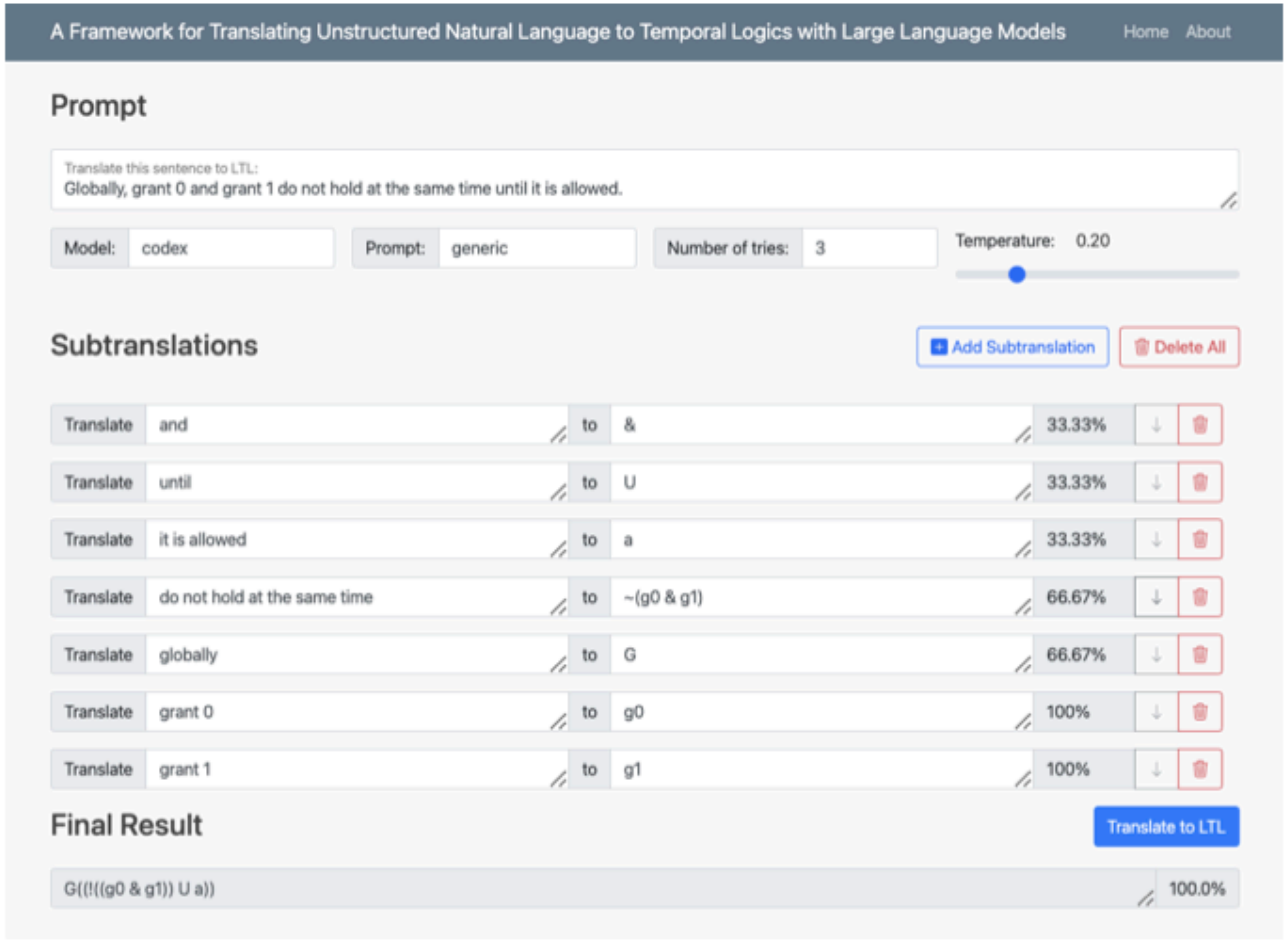
$\phi \mathcal{U} \psi$: “phi holds until psi”

$\Box \phi$: “phi always hold in the future"

$\diamond \phi$: “phi eventually will hold”

And standard first order logic

Cosler & al. CAV23


A Framework for Translating Unstructured Natural Language to Temporal Logics with Large Language Models

Home About

Prompt

Translate this sentence to LTL:
Globally, grant 0 and grant 1 do not hold at the same time until it is allowed.

Model: codex

Prompt: generic

Number of tries: 3

Temperature: 0.20

Subtranslations

Add Subtranslation

Delete All

Translate	and	to	&	33.33%
Translate	until	to	U	33.33%
Translate	it is allowed	to	a	33.33%
Translate	do not hold at the same time	to	~(g0 & g1)	66.67%
Translate	globally	to	G	66.67%
Translate	grant 0	to	g0	100%
Translate	grant 1	to	g1	100%

Final Result

Translate to LTL

G((!((g0 & g1)) U a)) 100.0%

Fig. 1: A screenshot of the web-interface for `nl2spec`.

What is it not so good for?

Even with temporal logic

How do I prove my Matrix Multiply accelerator is correct?

What's P (doable convincingly), what's Inv (k-induction will go to hell) ?

How do I prove that my RISCV processor is correct?

What's P, what's I?

What is correct anyway?

Hope of a more promising approach

Specifying externally

First proposal:

$(N, I, O, r, u, o) \prec (N', I, O, r', u', o')$

For all initial state i_1 of the the first machine,

For all input trace ,

There exists an initial state of the second machine i_2 *such that*

Applying the sequence of input will to the two machines starting in the two respective states will generate the same sequence of outputs

Morally this is IO trace inclusion

Traditional simulation relations

We say that a relation R $\subset \mathbb{B}^N \times \mathbb{B}^{N'}$ is a simulation relation if:

$$\forall\ l\ (p, q) \in R, \forall p', p \rightarrow_l p' \Rightarrow \exists q', q \rightarrow_l q', (p', q') \in R$$

If there exists a simulation such that $\forall (p, q) \in R, o(p) = o'(q)$

then $(N, I, O, r, u, o) \prec (N', I, O, r', u', o')$

Exercise: Does trace inclusion implies the existence of a simulation relation?

What is this rigid simulation good for

Readability vs circuit quality between spec and implementation

(Similar to the case for combinational ckts)

Area/Frequency *at Architectural performance constant*

Can help with state space explosion problem (next slide)

Exploiting symmetries

The queue example

Implementation:

circular buffer

enqueuePointer

dequeuePointer

+ do the thing

Specification:

linear buffer

length

On enqueue (if length < buffer_length):

Copy all the inflight data one slot to the wri

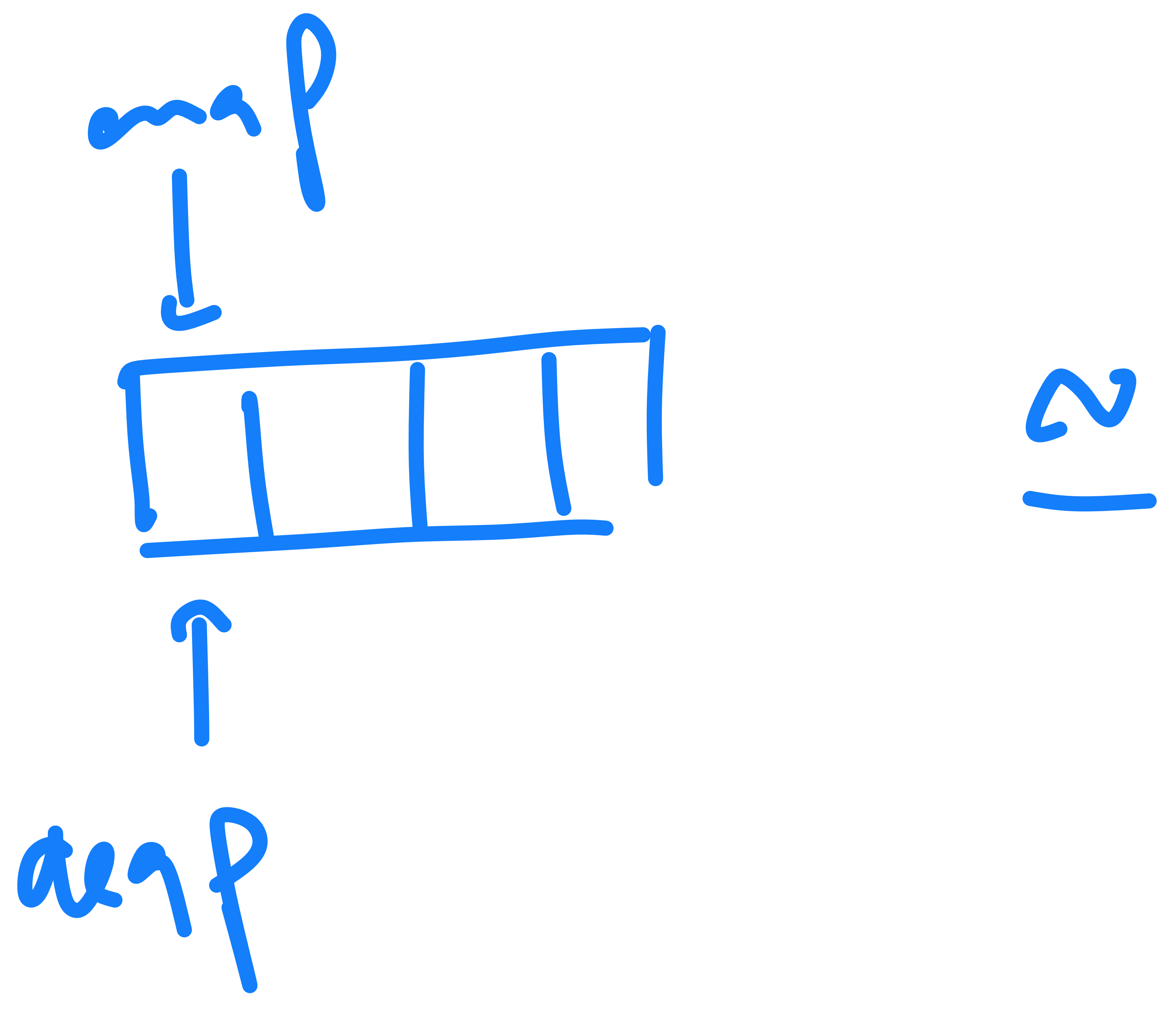
Add the new element at the beginning

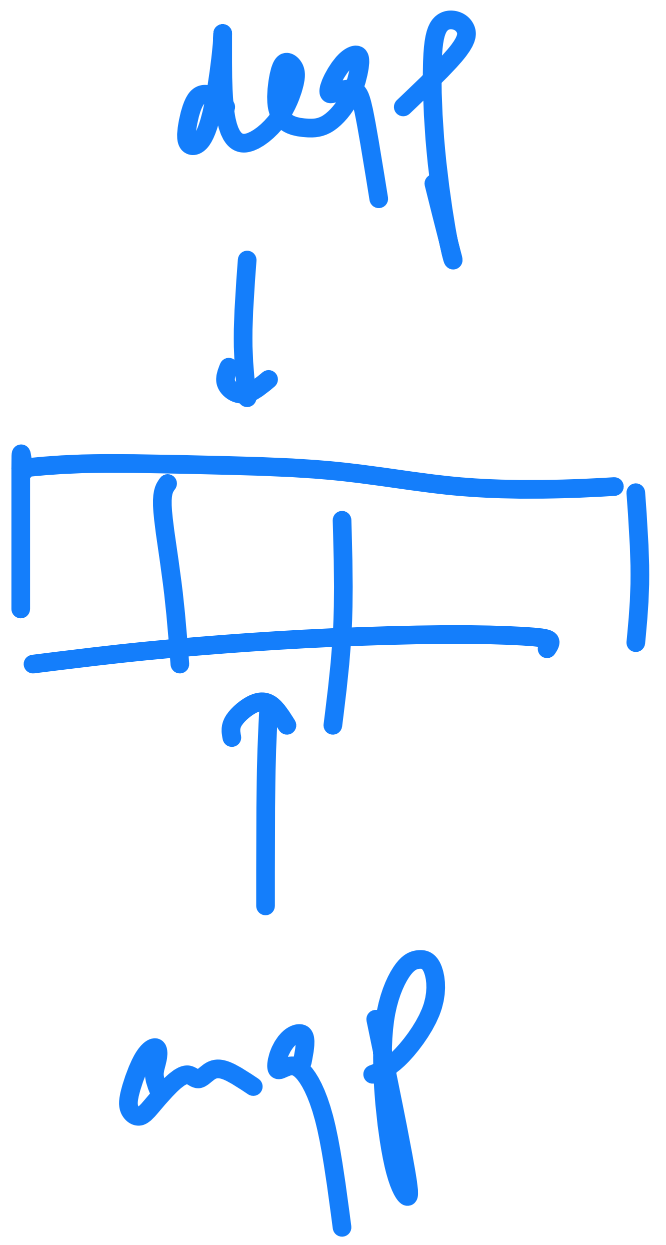
Many “indistinguishable” implementation state, are not bitwise equal!

In the spec equivalent states are bitwise equal!

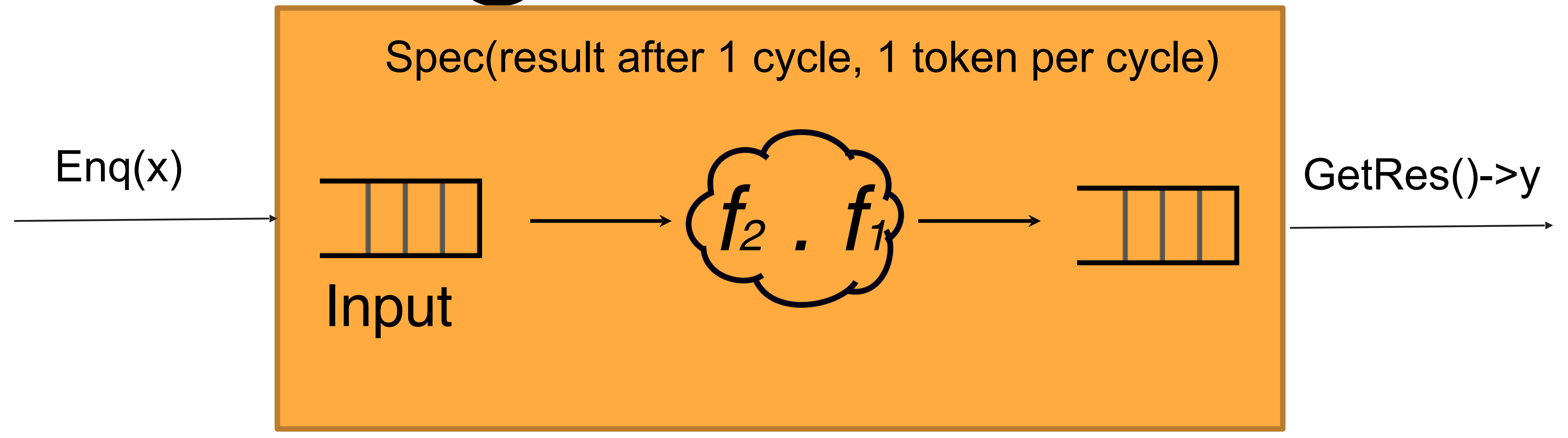
Solver without invariants won’t realise it reexplores “equivalent scenarios”

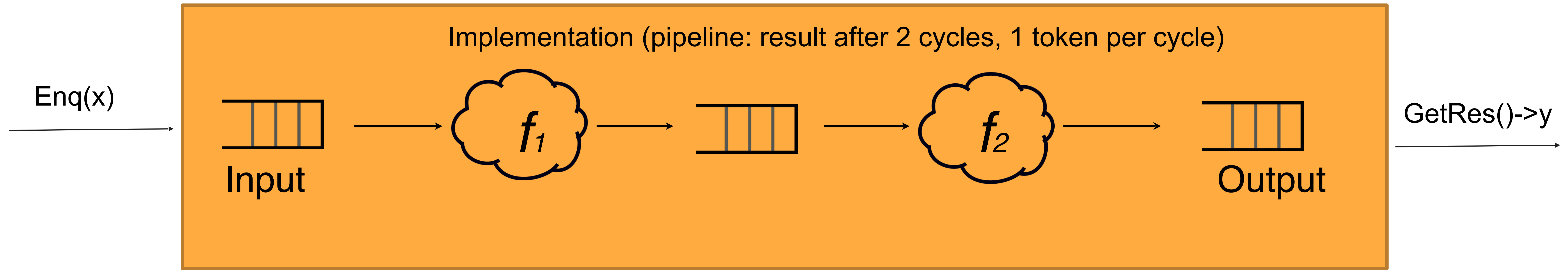
Slide to draw explanations about queue symmetries





What is it not so good for




Implementation (pipeline: result after 2 cycles, 1 token per cycle)
Enq(x)
Input
f_1
f_2
Output
GetRes()->y

There is NO simulation/trace inclusion!

Simplest solution: recyling a variation of simulation idea, as a DUT!

-> *Encoding* a custom notion of equivalence inside the DUT which would do differential testing

Taking a step back

We are married to the notion of clock cycle and it is bringing trouble:

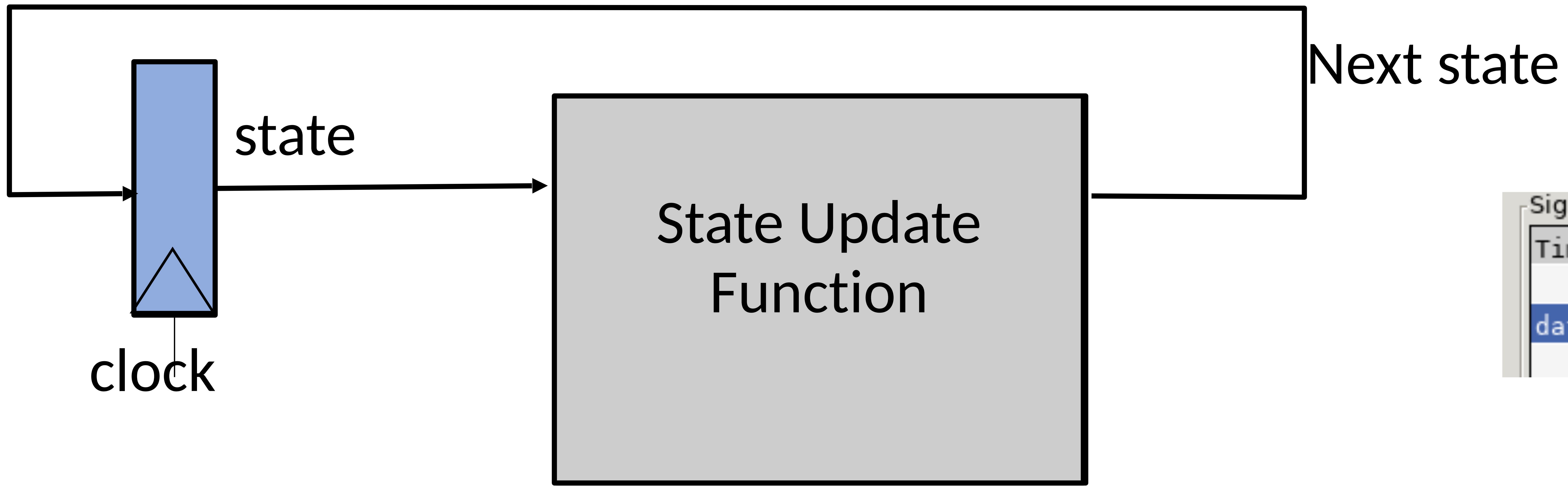
- Natural equivalence is too strong
- A cycle has lot of things that concurrently happens which are entangled together

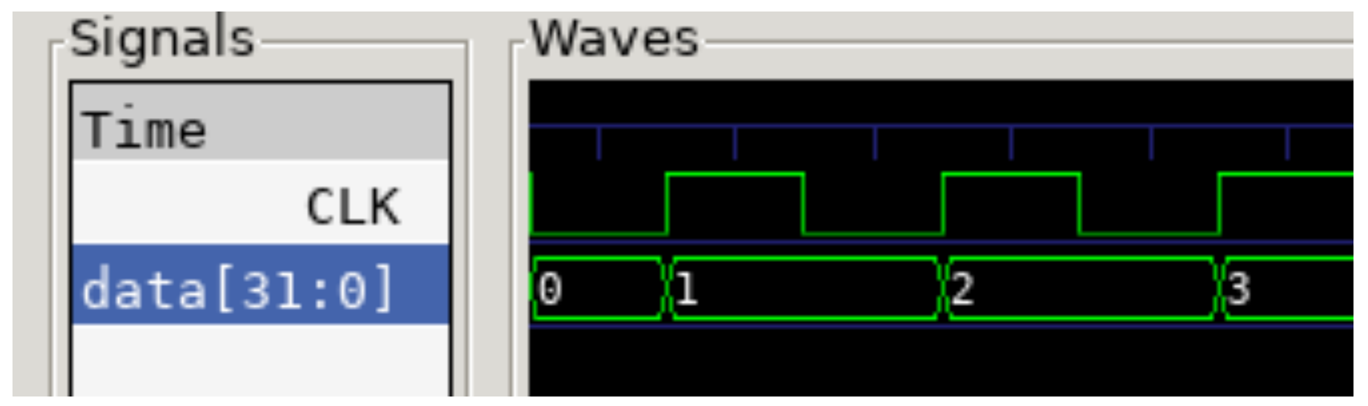
Desiderata:

- Reasonably flexible notion of circuit equivalence
- A way to decouple cycles from logical steps
- How to find/describe the Inv conveniently, when k-induction is not enough?

Part II:
Boolean Functions -> Moore’s Machine -> (? <-> Computer Architect Designs)

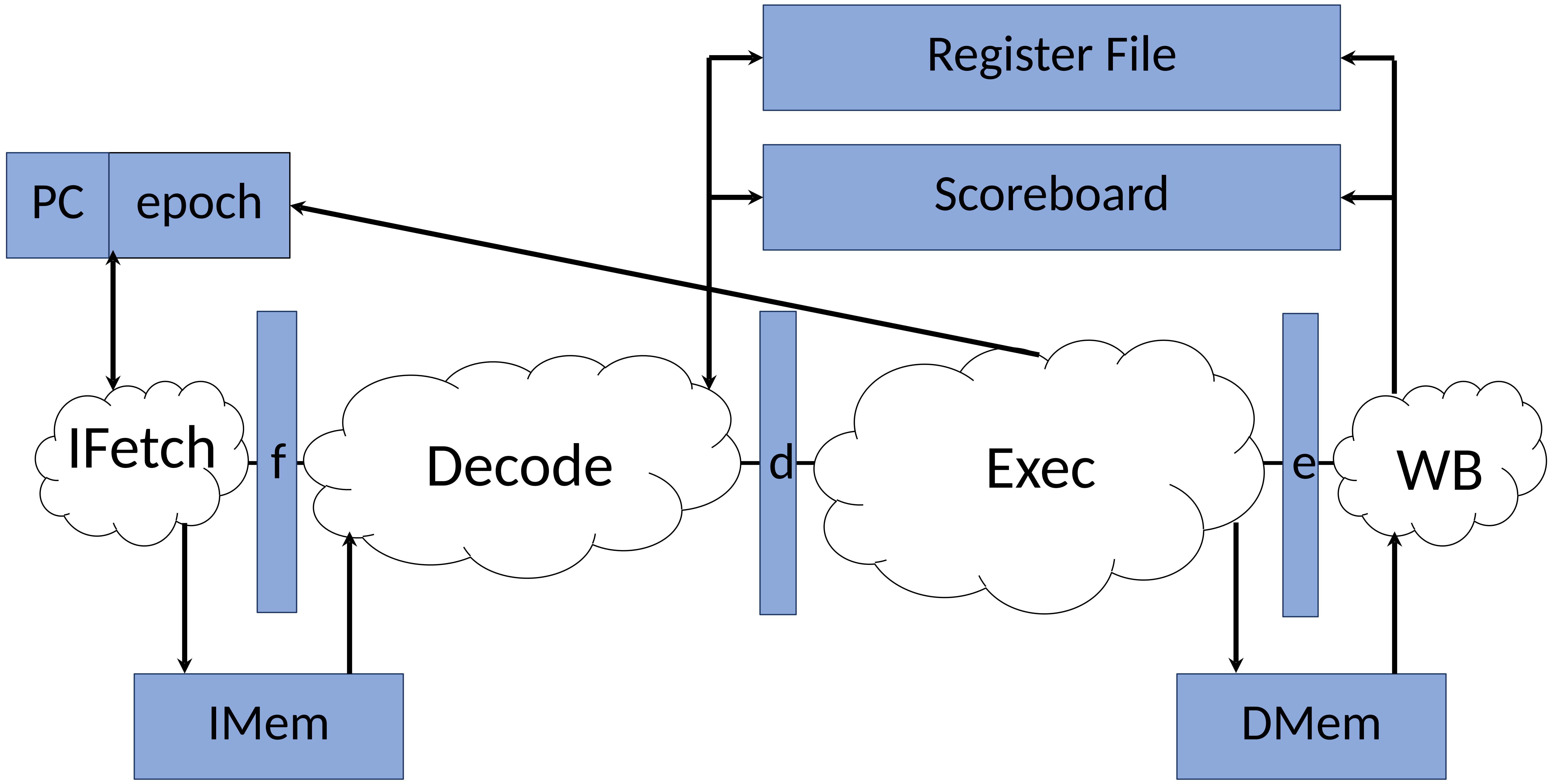
A view of circuits





Fixed number of registers
Only one new value per cycle

Cycles are the wrong abstraction

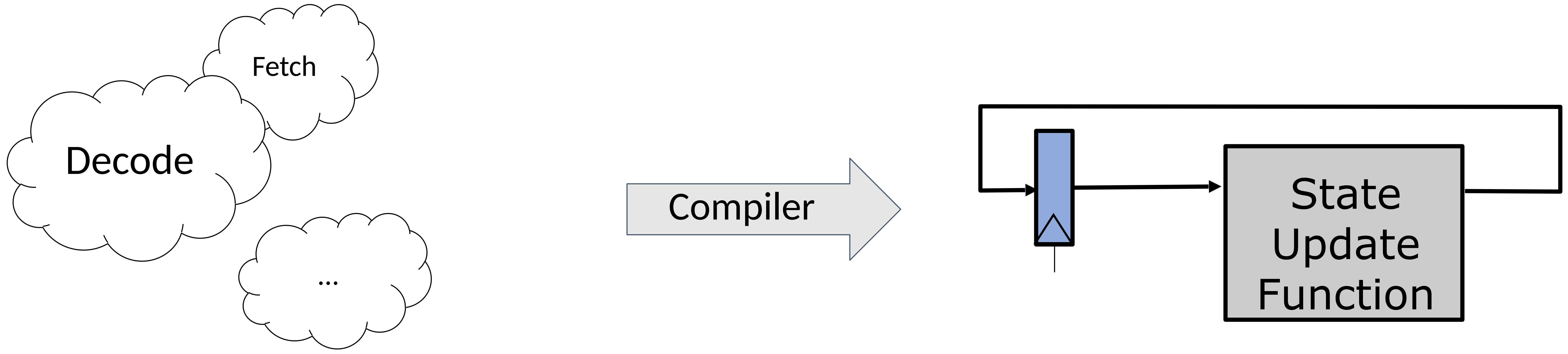


1 cycle does not have simple architectural meaning.

Sometimes it just does Fetch.

Sometimes it just does Execute… sometimes it does everything!

Bluespec: The rule-based methodology

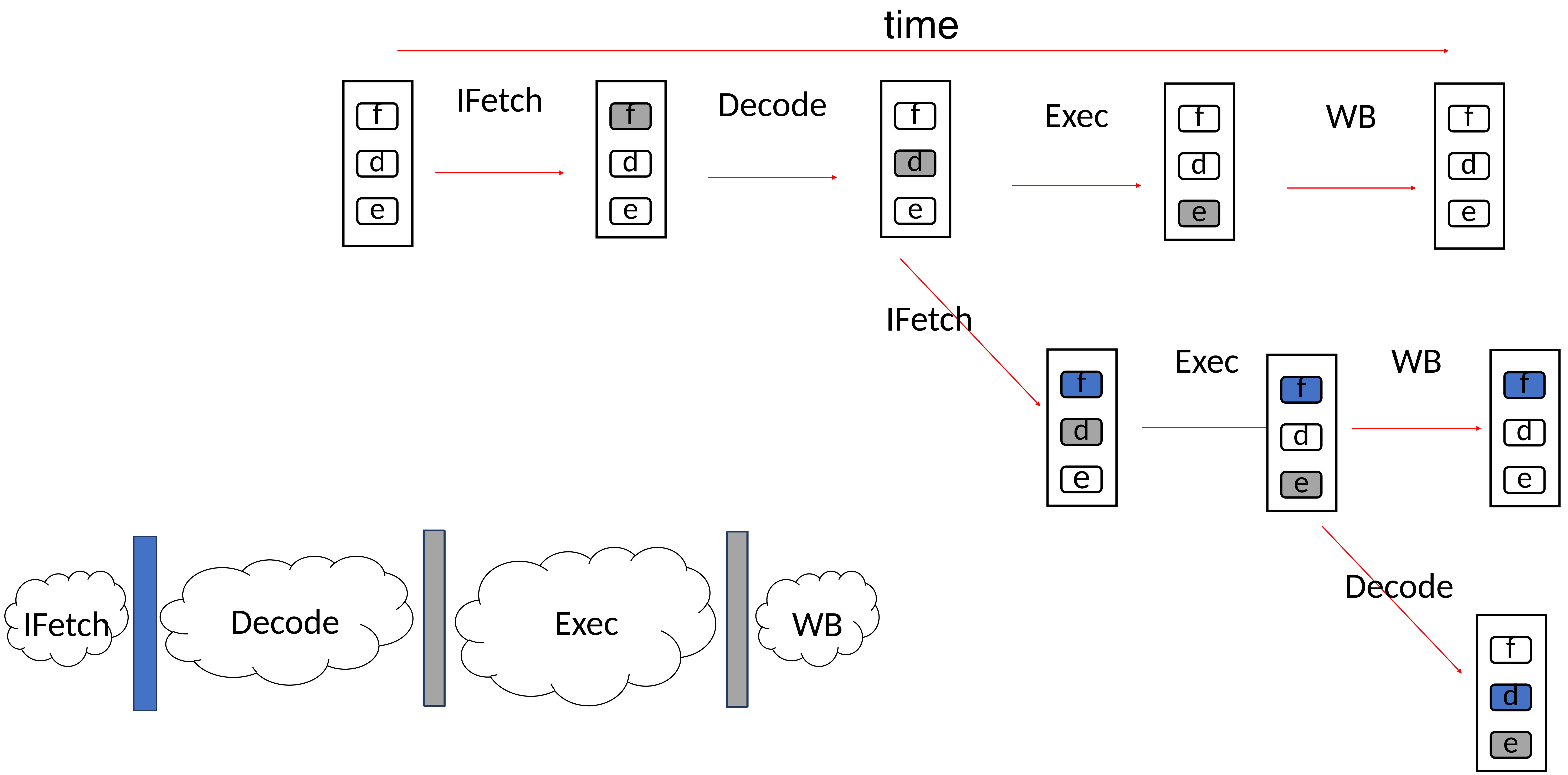


Small steps, named rules, are atomic (conceptually combinational)

The compiler assembles those rules to build the state update function

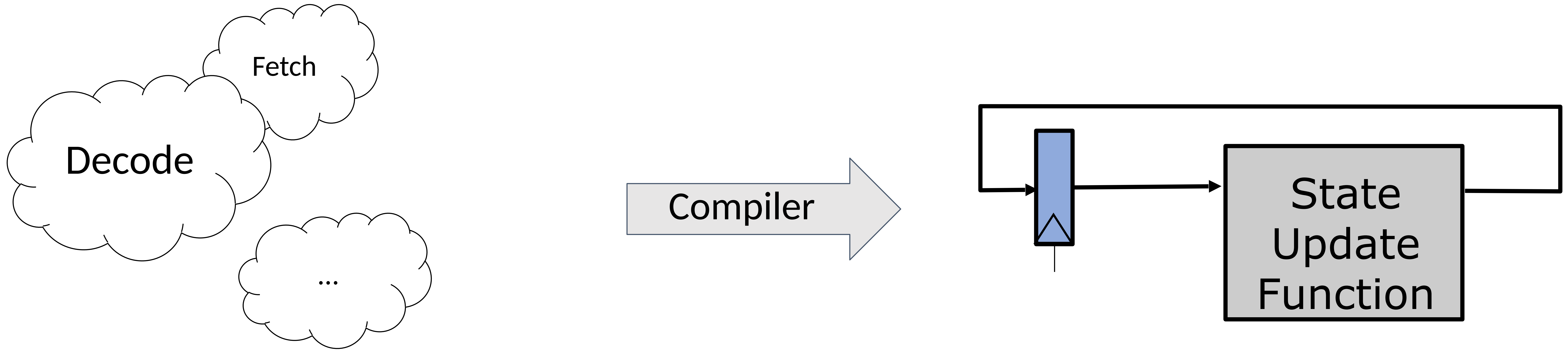
It is a nonobjective that every Sequential circuit be expressible as a Bluespec program

One rule at a time



/!\ We studied the processor “unit by unit”, and harnessed concurrency, but we lost control of time !

Bluespec: The rule-based methodology



Rules, are epsilon transitions

Interaction with modules are through methods (what is it for our processor? maybe two special “put” and “get" instructions)

Modules

Processor, Queue, Register file, Reorder buffer, Memory, etc…

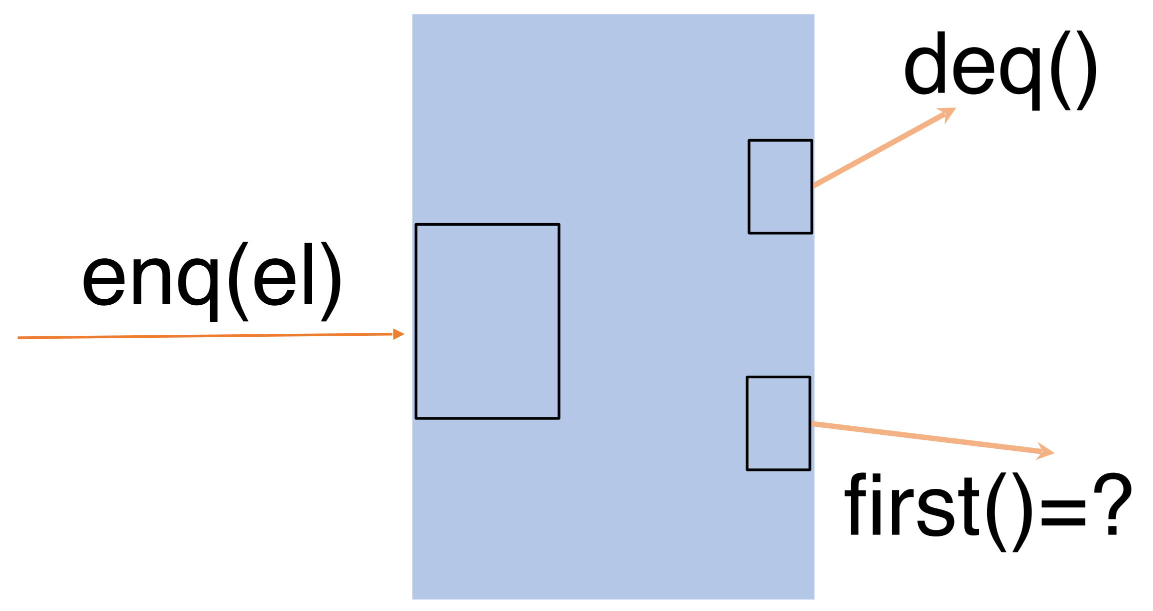
Interacting through ready/enable

- Guards: sometimes one cannot call a method, or a rule cannot execute

Example queue:

- Rules: _
- Action method (state change): enq, deq
- Value method (observation): first

Ideally: modules one transition at a time



Characterize a module with the collection of the transitions induced by its methods and by its rules, each considered in isolation

Modules are fully described by a few transitions

State: collection of submodule states: $S = S_1 \times \dots \times S_k$

Rules (Spontaneous transitions):

$r \subseteq S \times S$ (old state, new state)

Method relations

Action methods (partial stateful change):

am $\subseteq S \times \text{Value} \times S$

(old state, argument, new state)

Value methods (partial observation):

vm $\subseteq S \times \text{Value} \times \text{Value}$

(old state, argument, value returned)

Relations can be non-functional
(non-determinism)
Relations can be partial
(guards)

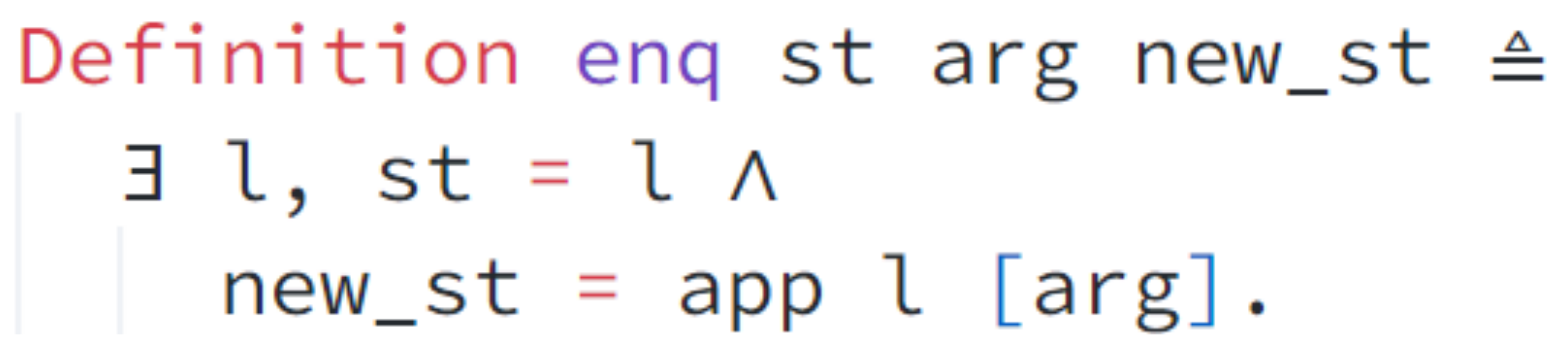
Multiple ways to define modules

Modules can be defined either:

from a Fjfj definition *(potentially synthesized to hardware)*

or directly as a logical proposition in Coq: *a mathematical model*

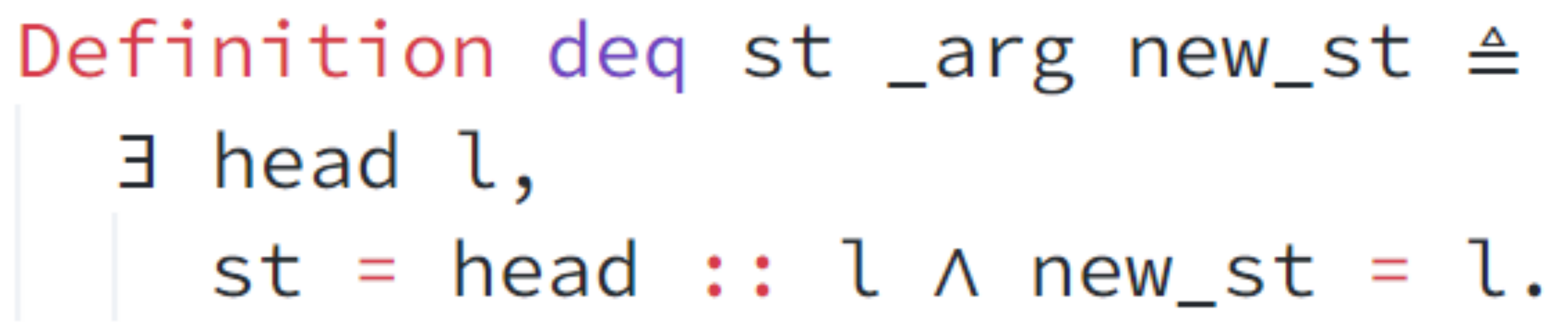
Mathematical models for modules



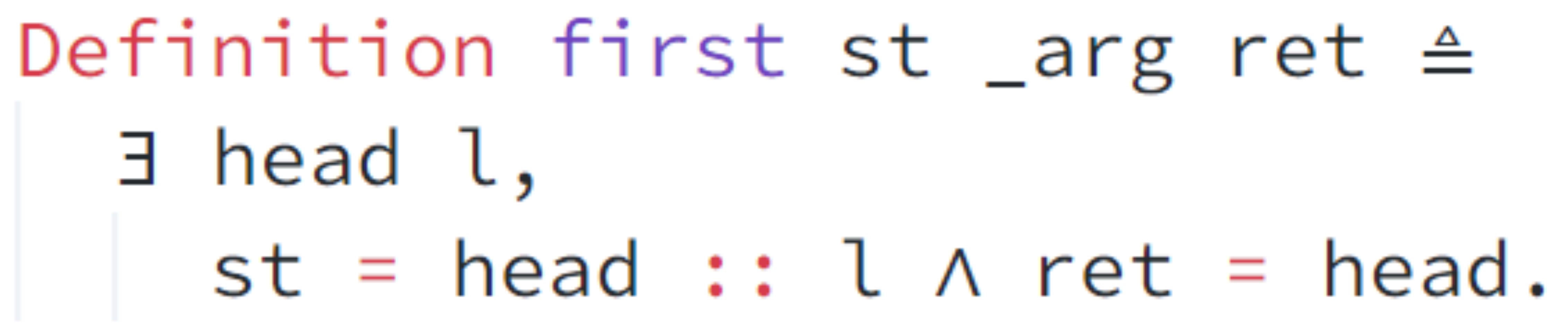
```
Definition enq st arg new_st ≜
  ∃ l, st = l ∧
    new_st = app l [arg].
```

enq $\subseteq S \times \text{Value} \times S$

enq = { (st, arg, new_st) | enq st arg new_st }



```
Definition deq st _arg new_st ≜
  ∃ head l,
    st = head :: l ∧ new_st = l.
```



```
Definition first st _arg ret ≜
  ∃ head l,
    st = head :: l ∧ ret = head.
```

Defining methods in Fjfj

```
Definition enq ≜
  (action_method (e)
    (begin
      (if (= {read valid} 0)
        (begin
          {write data e}
          {write valid 1})
        abort))).
```

```
Definition first ≜
  (value_method ()
      (if (= {read valid} 1)
          {read data}
          abort)).
```

Registers are modules too

```
Definition write _st arg new_st ≜
    new_st = arg.
Definition read st _arg ret ≜
    ret = st.
```

Very easy non-determinism

```
Definition flimsy_write st arg new_st ≜
    (new_st = arg) \/ (new_st = st).
```

```
Definition choose_between st arg ret ≜
    ∃ n,
      dlet {min max} ≜ arg in
      min <= n ∧
      n < max ∧
      ret = n.
```

Multiple transition relations (with epsilon transitions)

-> flexible notion of simulation

Simulation relations still work!

Minus small tweak

We say that a relation R $\subset S_1 \times S_2$ is a simulation relation if:

$$\forall \quad (p, q) \in R, \forall p', p \rightarrow p' \Rightarrow \exists q', q \rightarrow^* q', (p', q') \in R$$

With the “steps" relation being application of any of the rules of the system

AND

We need similar preservation of simulation for the action methods

The value methods are the collection of observations one can do on the state)

Simulation diagrams

$$\begin{tikzcd}
i1 \arrow[r, dash] \arrow[dddd, "r"'] & s1 \arrow[d, dashed, "r1"] \\
 & s2 \arrow[d, dashed, "r2"] \\
 & s3 \arrow[d, dotted, dash] \\
 & \vdots \arrow[d, dashed, "rk"] \\
i2 \arrow[r, dashed, dash] & sk
\end{tikzcd}$$

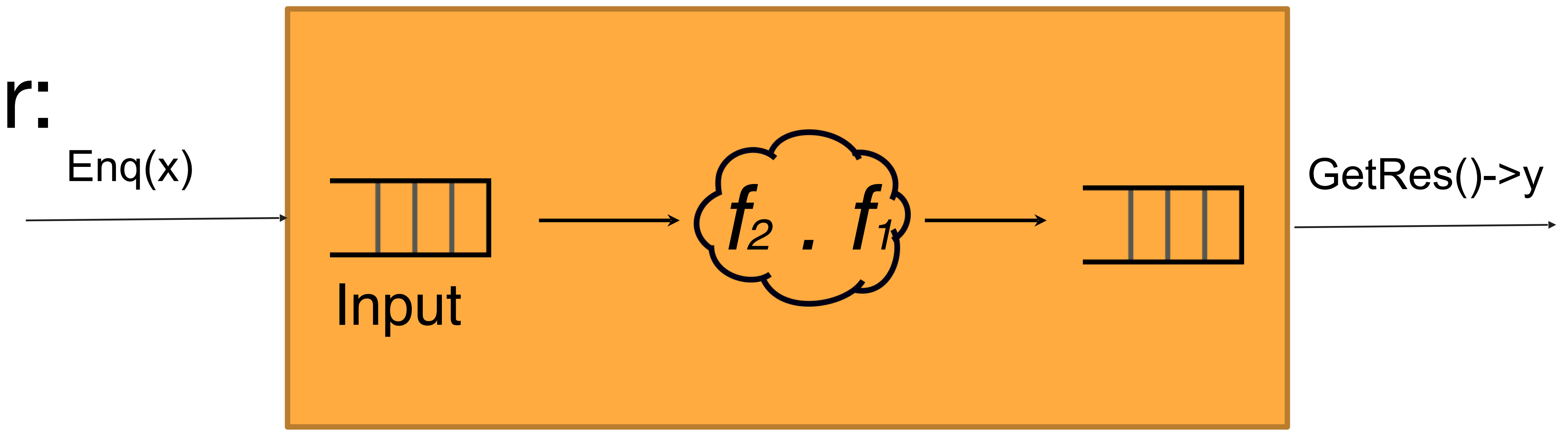
$$\begin{tikzcd}
i1 \arrow[r, dash] \arrow[dddd, "am(arg)"'] & s1 \arrow[d, "am(arg)"] \\
 & s2 \arrow[d, dashed, "r2"] \\
 & s3 \arrow[d, dotted, dash] \\
 & \vdots \arrow[d, dashed, "rk"] \\
i2 \arrow[r, dashed, dash] & sk
\end{tikzcd}$$

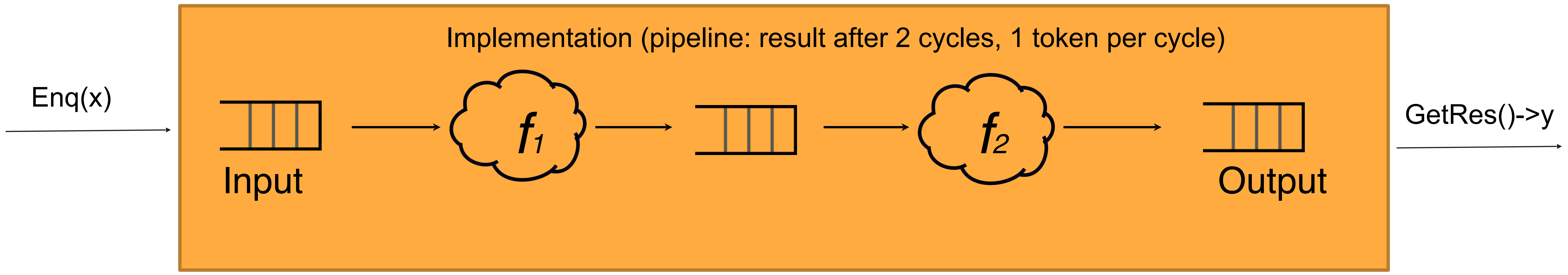
Stepping through in Coq to do very easy implementation and simulation proof

Frustration with this proof

We had to manually give the simulation relation :(

Exercise, what is R for:





Implementation (pipeline: result after 2 cycles, 1 token per cycle)
Enq(x)
Input
f_1
f_2
Output
GetRes()->y

Intuition: Burch & Dill CAV 94

“Flushing" gives (often) a simulation relation! (Drawing Area)

Encoding flushing: manually and inductive propositions! Back to Coq/
Rosette

K-induction in the multi-transition case

An idea I want to explore

With just two rules, a 2-induction is:

```
P init -> (forall x, rule1 init x -> P x) -> (forall x, rule2 init x -> P x) ->
  (Forall x x’ x”, P x -> rule1 x  x’ -> P x’ -> rule1 x’ x” -> P x”) ->
  (Forall x x’ x”, P x -> rule1 x  x’ -> P x’ -> rule2 x’ x” -> P x”) ->
  (Forall x x’ x”, P x -> rule2 x  x’ -> P x’ -> rule1 x’ x” -> P x”) ->
  (Forall x x’ x”, P x -> rule2 x  x’ -> P x’ -> rule2 x’ x” -> P x”) ->
  forall x reachable, P x
```

Let’s do a custom unbalanced induction principle:

Lazily expand with more history, if one of the inductive hypothesis does not pass in itself

Conclusion

From Boolean functions, to Moore’s Machine, to multi-transition systems. There are easy way to denote (give semantics) to large class of synchronous circuits.

We saw standard methodology, where verification and design are intertwined, standard equivalence and there limitation

Introduced our approach, escaping from cycles, embedding the multitransition systems in Coq: fishing simulations flushing tricks and k-induction