UNIVERSITY OF BIRMINGHAM

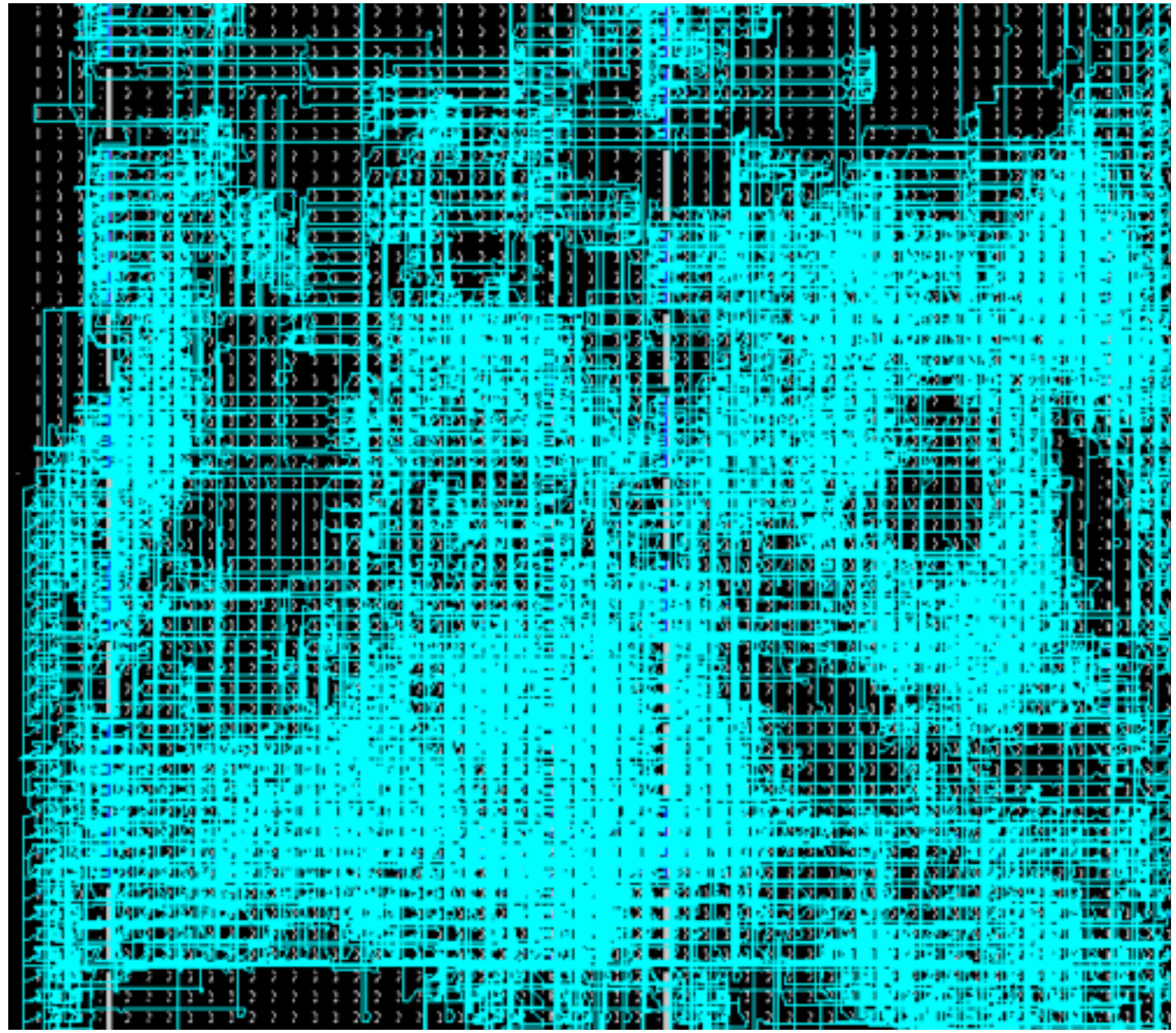
Generating hardware from game semantics

Dan R. Ghica

UNIVERSITY OF BIRMINGHAM

Generating hardware from game semantics

Dan R. Ghica



from (programming) languages to circuits

basic *syntactic control of interference*

$$\dfrac{}{x : \theta \vdash x : \theta}\ \text{Identity} \qquad \dfrac{\Gamma \vdash M : \theta}{\Gamma, x : \theta' \vdash M : \theta}\ \text{Weakening}$$

$$\dfrac{\Gamma, x : \theta' \vdash M : \theta}{\Gamma \vdash \lambda x.M : \theta' \to \theta} \to \text{Introduction}$$

$$\dfrac{\Gamma \vdash F : \theta' \to \theta \qquad \Delta \vdash M : \theta'}{\Gamma, \Delta \vdash FM : \theta} \to \text{Elimination}$$

$$\dfrac{\Gamma \vdash M : \theta' \qquad \Gamma \vdash N : \theta}{\Gamma \vdash \langle M, N \rangle : \theta' \times \theta} \times \text{Introduction}$$

basic *syntactic control of interference*

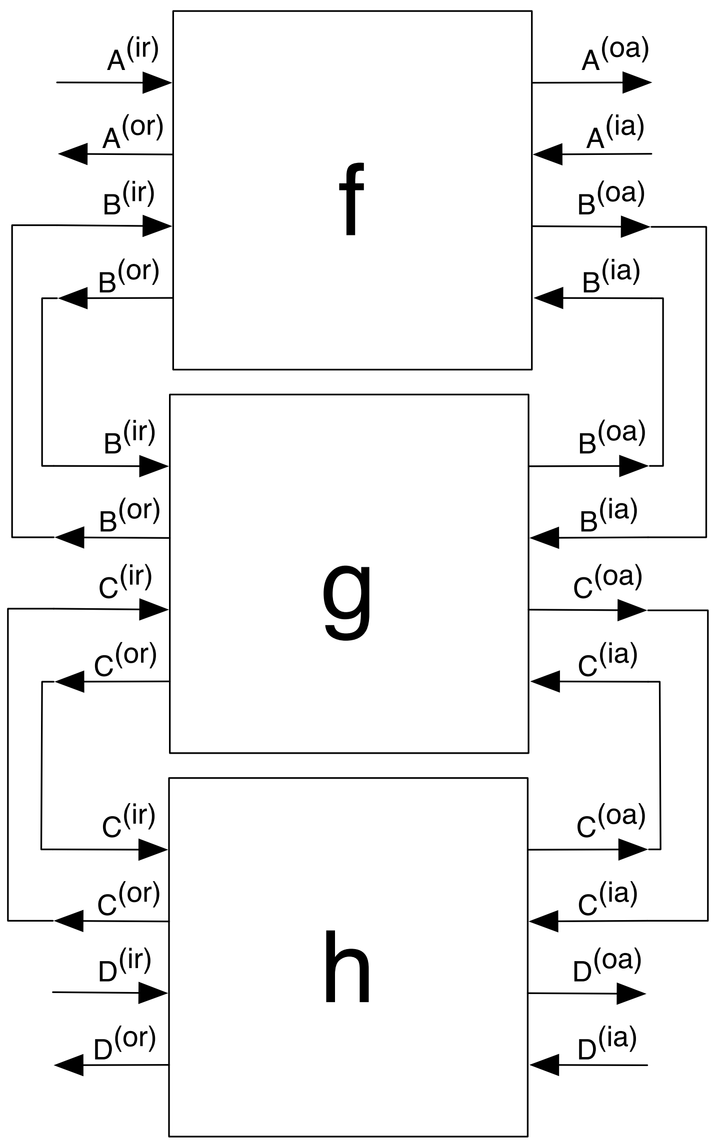
$1 : \mathsf{exp}$	constant
$0 : \mathsf{exp}$	constant
$\mathsf{skip} : \mathsf{com}$	no-op
$\mathsf{asg} : \mathsf{cell} \times \mathsf{exp} \to \mathsf{com}$	assignment
$\mathsf{der} : \mathsf{cell} \to \mathsf{exp}$	dereferencing
$\mathsf{seq} : \mathsf{com} \times \mathsf{com} \to \mathsf{com}$	sequencing
$\mathsf{seq} : \mathsf{com} \times \mathsf{exp} \to \mathsf{exp}$	sequencing with boolean
$\mathsf{op} : \mathsf{exp} \times \mathsf{exp} \to \mathsf{exp}$	logical operations
$\mathsf{if} : \mathsf{exp} \times \mathsf{com} \times \mathsf{com} \to \mathsf{com}$	branching
$\mathsf{while} : \mathsf{exp} \times \mathsf{com} \to \mathsf{com}$	iteration
$\mathsf{newvar} : (\mathsf{cell} \to \mathsf{com}) \to \mathsf{com}$	local variable
$\mathsf{newvar} : (\mathsf{cell} \to \mathsf{exp}) \to \mathsf{exp}$	local variable.

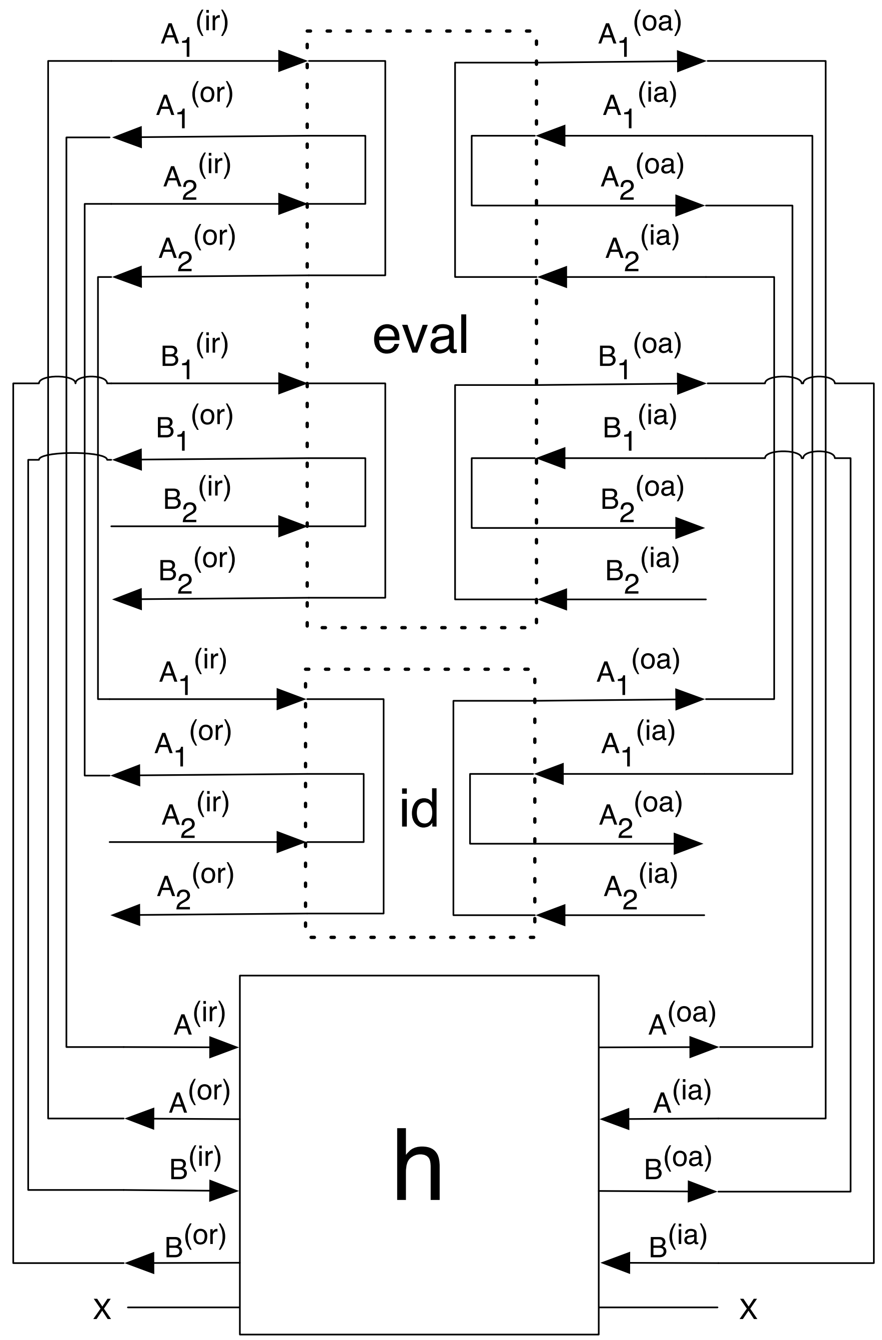
basic *syntactic control of interference*

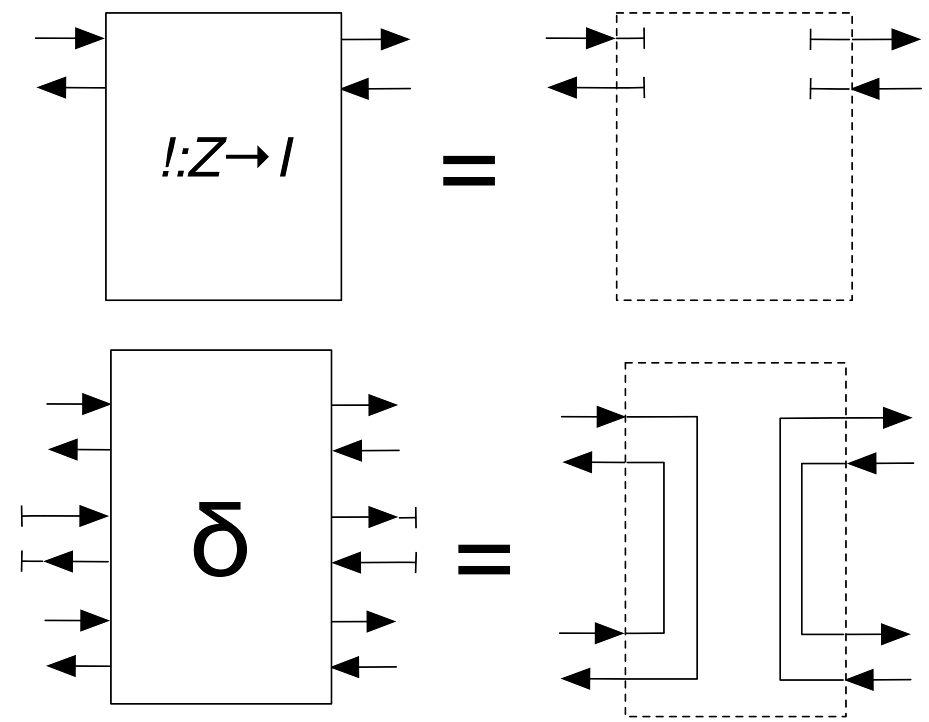
$1 : \mathsf{exp}$	constant
$0 : \mathsf{exp}$	constant
$\mathsf{skip} : \mathsf{com}$	no-op
$\mathsf{asg} : \mathsf{cell} \times \mathsf{exp} \to \mathsf{com}$	assignment
$\mathsf{der} : \mathsf{cell} \to \mathsf{exp}$	dereferencing
$\mathsf{seq} : \mathsf{com} \times \mathsf{com} \to \mathsf{com}$	sequencing
$\mathsf{seq} : \mathsf{com} \times \mathsf{exp} \to \mathsf{exp}$	sequencing with boolean
$\mathsf{op} : \mathsf{exp} \times \mathsf{exp} \to \mathsf{exp}$	logical operations
$\mathsf{if} : \mathsf{exp} \times \mathsf{com} \times \mathsf{com} \to \mathsf{com}$	branching
$\mathsf{while} : \mathsf{exp} \times \mathsf{com} \to \mathsf{com}$	iteration
$\mathsf{newvar} : (\mathsf{cell} \to \mathsf{com}) \to \mathsf{com}$	local variable
$\mathsf{newvar} : (\mathsf{cell} \to \mathsf{exp}) \to \mathsf{exp}$	local variable.

$$\boxed{\mathsf{par} : \mathsf{com} \to \mathsf{com} \to \mathsf{com}.}$$

geometry of synthesis: a "direct" circuit semantics







closed monoidal category with cartesian products

$$\llbracket x:\theta \vdash x:\theta \rrbracket = id_{\llbracket \theta \rrbracket}$$

$$\llbracket \Gamma, x:\theta' \vdash M:\theta \rrbracket = \llbracket \Gamma \vdash M:\theta \rrbracket \circ \pi_1$$

$$\llbracket \Gamma \vdash \lambda x.M:\theta' \to \theta \rrbracket = \Lambda(\llbracket \Gamma, x:\theta' \vdash M:\theta \rrbracket)$$

$$\llbracket \Gamma, \Delta \vdash FM:\theta \rrbracket = \mathrm{eval} \circ \left(\llbracket \Delta \vdash M:\theta' \rrbracket \otimes \llbracket \Gamma \vdash F:\theta' \to \theta \rrbracket \right)$$

$$\llbracket \Gamma \vdash \langle M, N \rangle : \theta \times \theta' \rrbracket = \left(\llbracket \Gamma \vdash M:\theta \rrbracket \otimes \llbracket \rho(\Gamma \vdash N:\theta') \rrbracket \right) \circ \delta_{\llbracket \Gamma \rrbracket},$$

closed monoidal category with cartesian products

$$[\![x:\theta \vdash x:\theta]\!] = id_{[\![\theta]\!]}$$

$$[\![\Gamma, x:\theta' \vdash M:\theta]\!] = [\![\Gamma \vdash M:\theta]\!] \circ \pi_1$$

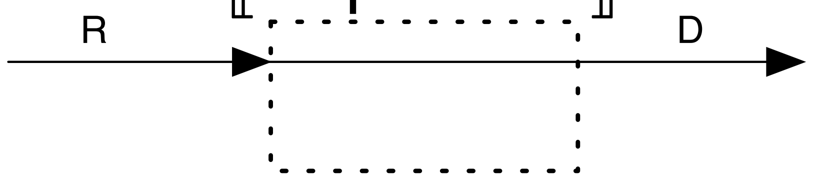
$$[\![\Gamma \vdash \lambda x.M:\theta' \to \theta]\!] = \Lambda([\![\Gamma, x:\theta' \vdash M:\theta]\!])$$

$$[\![\Gamma, \Delta \vdash FM:\theta]\!] = \mathrm{eval} \circ \left([\![\Delta \vdash M{:}\theta']\!] \otimes [\![\Gamma \vdash F{:}\theta'{\to}\theta]\!]\right)$$

$$\boxed{[\![\Gamma \vdash \langle M, N\rangle : \theta \times \theta']\!] = \left([\![\Gamma \vdash M{:}\theta]\!] \otimes [\![\rho(\Gamma \vdash N{:}\theta')]\!]\right) \circ \delta_{[\![\Gamma]\!]}},$$

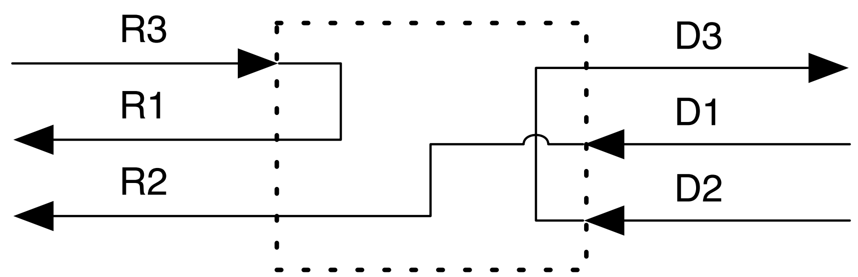
game-inspired semantics for language constants

$[\![\mathsf{skip} : \mathsf{com}]\!]$



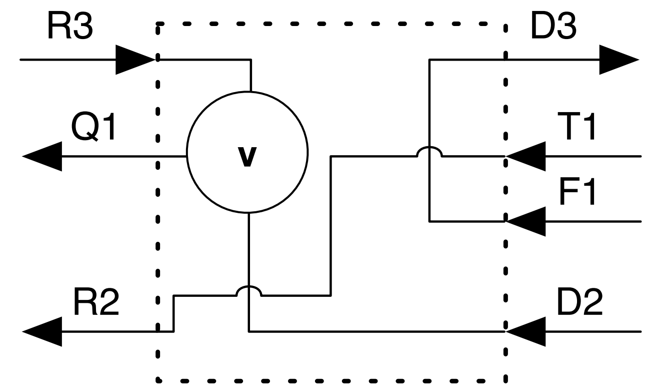
R.D

$[\![\mathsf{seq} : (\mathsf{com}_1 \times \mathsf{com}_2) \rightarrow \mathsf{com}_3]\!]$



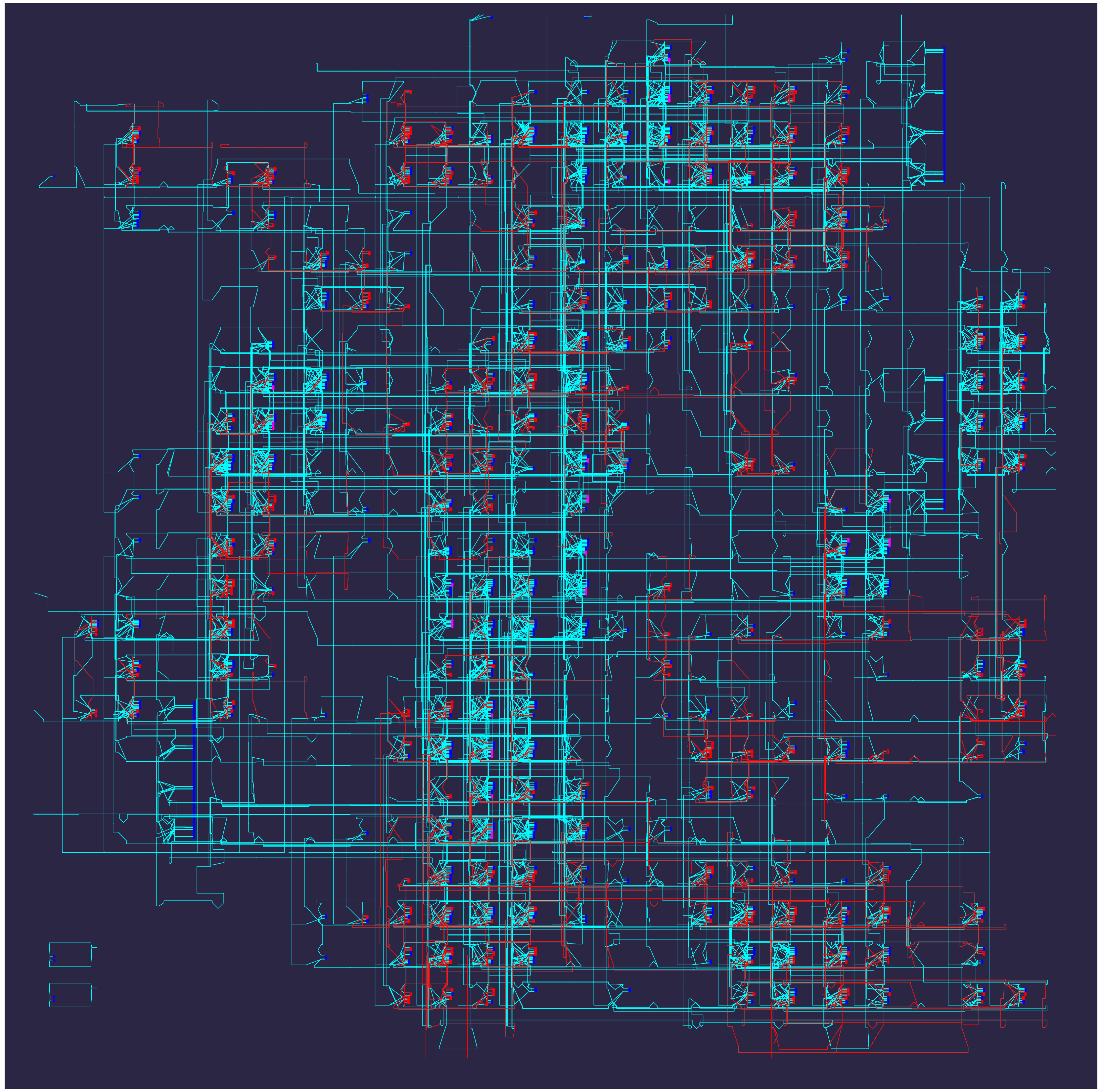
R3.R1.D1.R2.D2.D3

$[\![\mathsf{while} : (\mathsf{exp}_1 \times \mathsf{com}_2) \rightarrow \mathsf{com}_3]\!]$



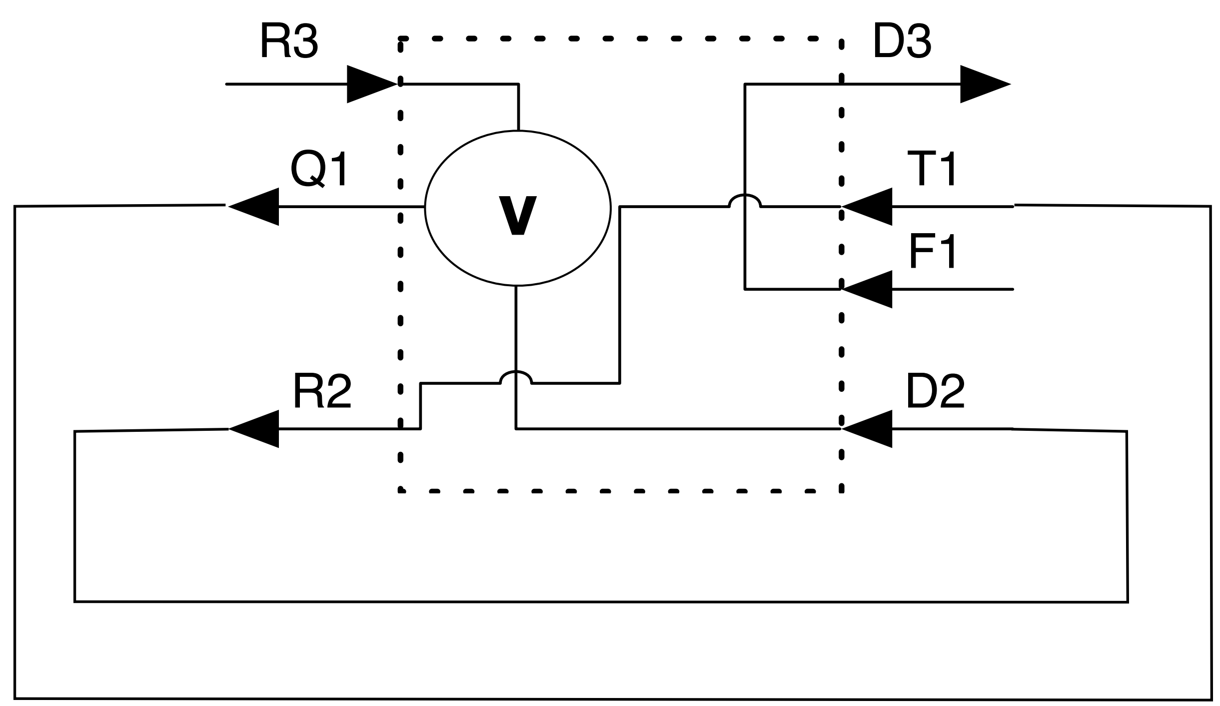
R3.Q1.T1.R2.D2.Q1.F.D3

it works



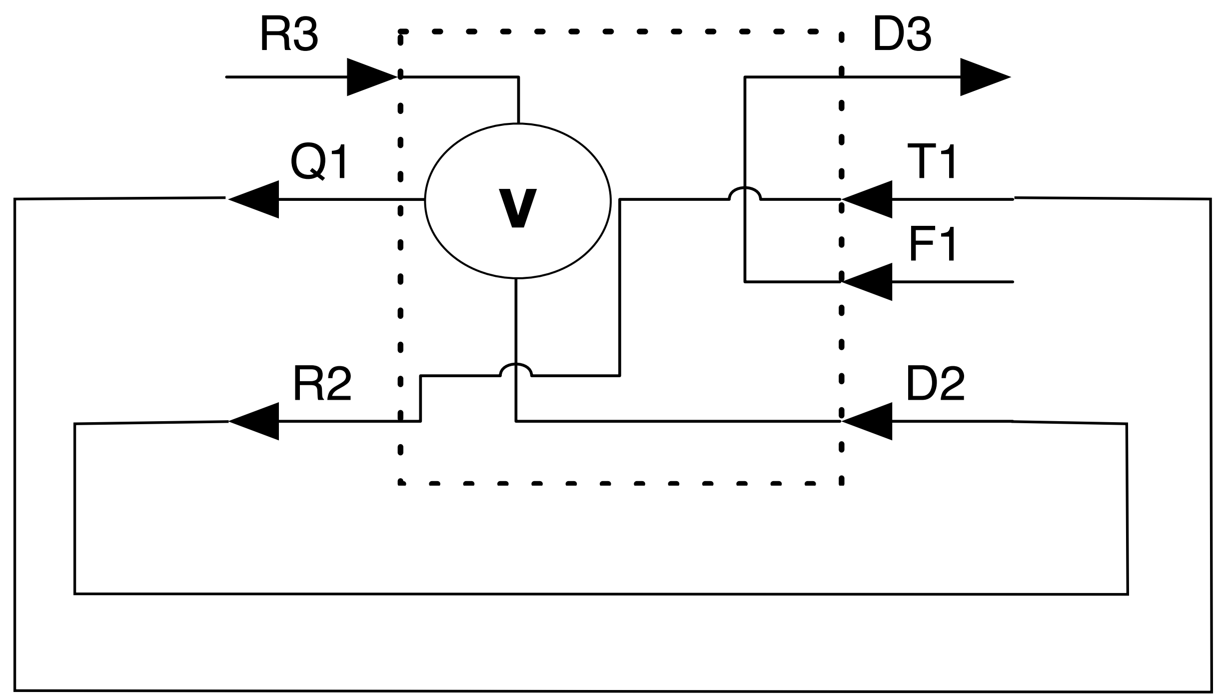
... but not as well as it should

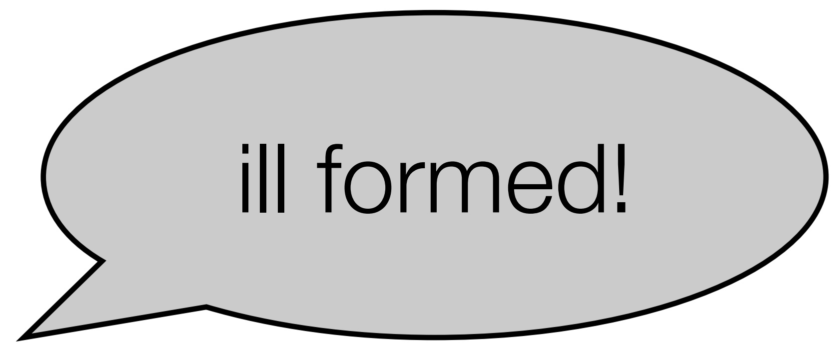
```
while true do skip
```



... but not as well as it should

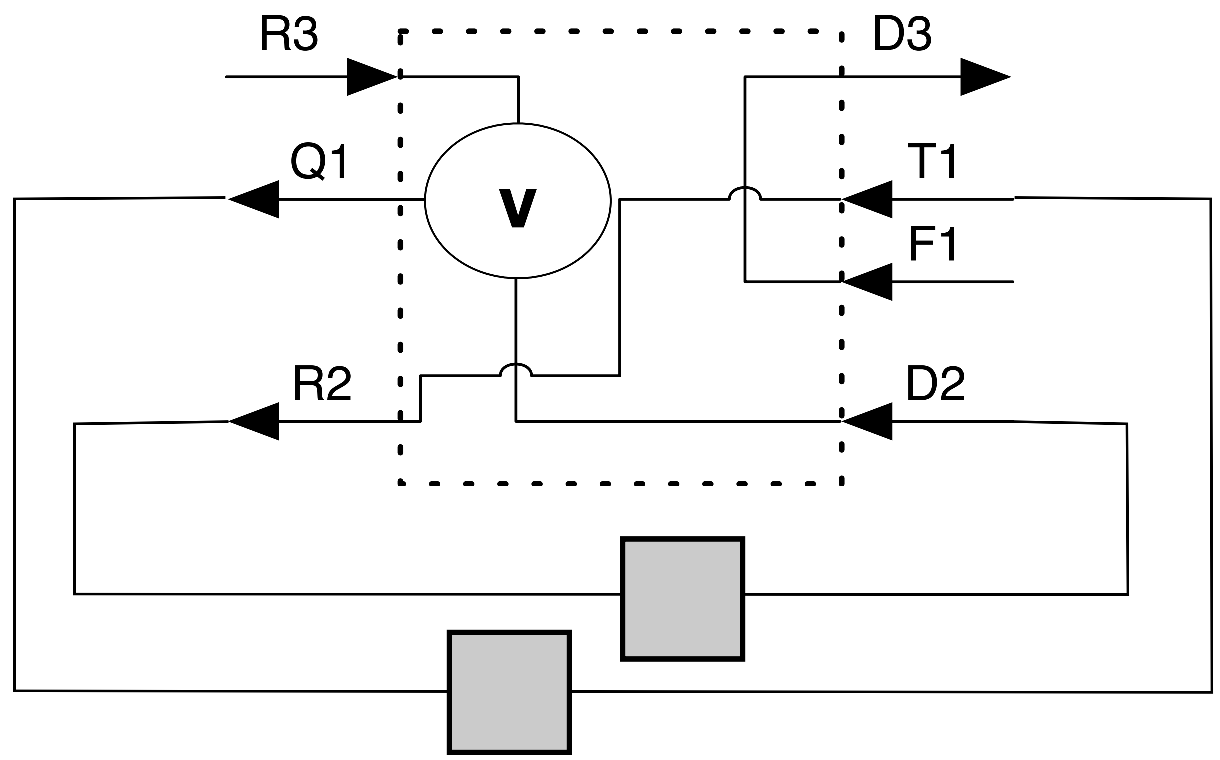
```
while true do skip
```

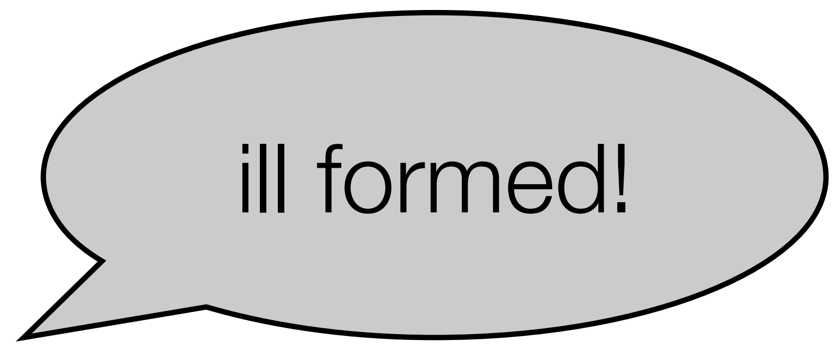




... but not as well as it should

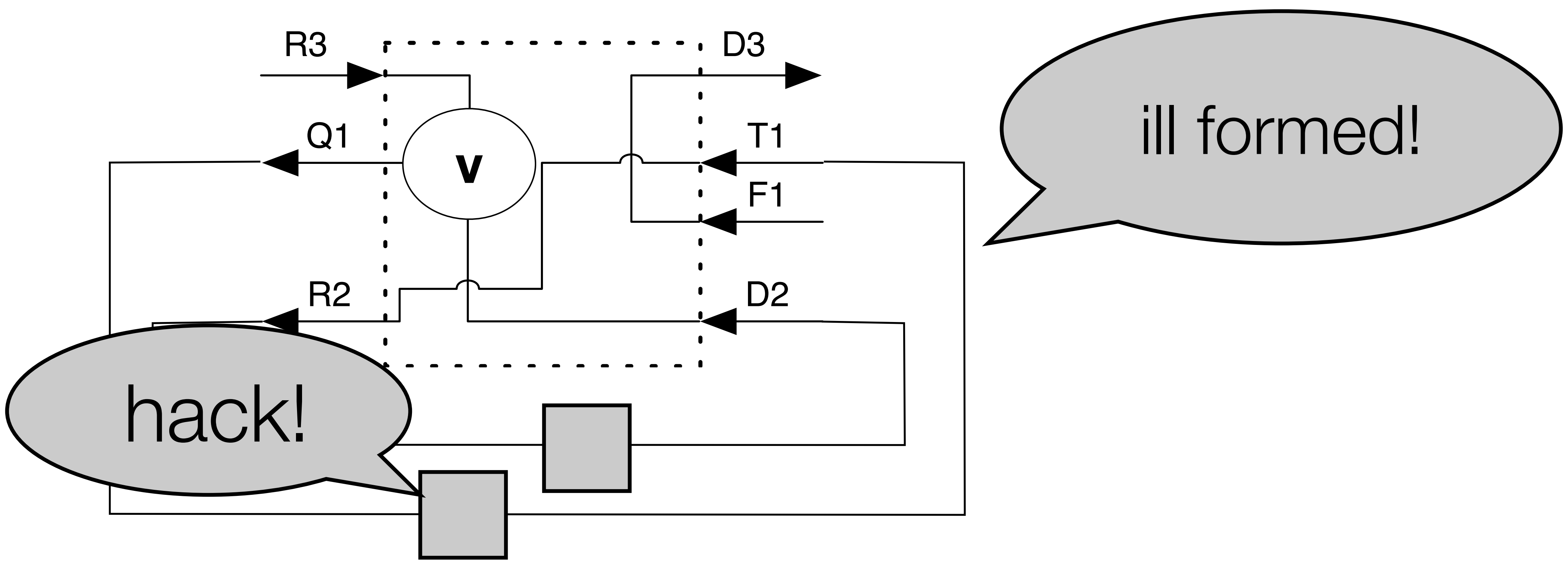
```
while true do skip
```





... but not as well as it should

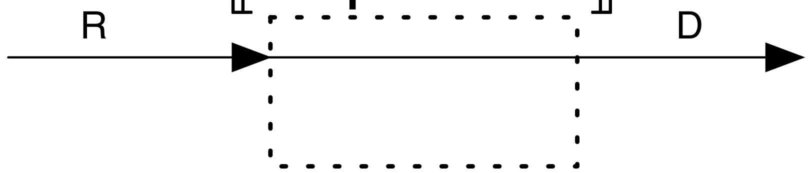
```
while true do skip
```



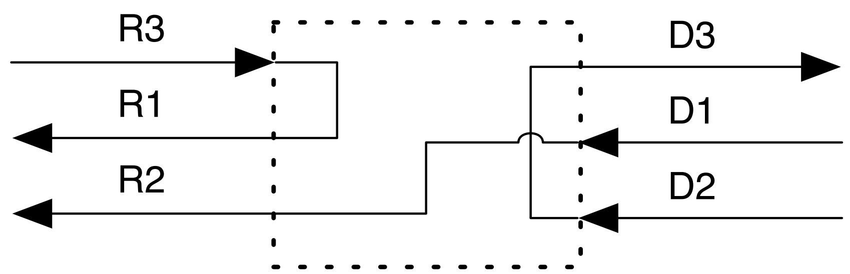
digital (clocked) hardware is synchronous
game-semantic models are asynchronous

synchronous traces for constants

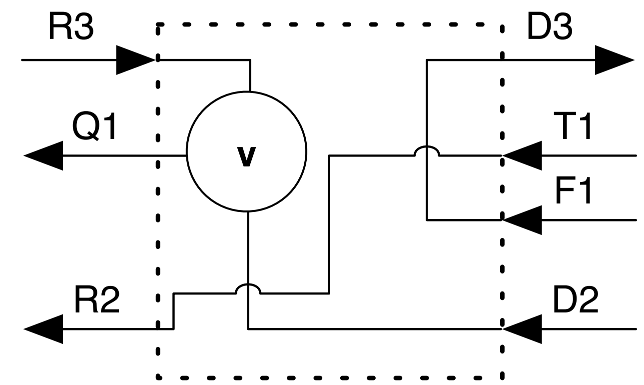
$[\![\mathsf{skip} : \mathsf{com}]\!]$



$[\![\mathsf{seq} : (\mathsf{com}_1 \times \mathsf{com}_2) \longrightarrow \mathsf{com}_3]\!]$

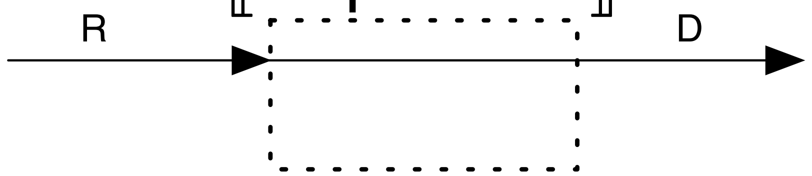


$[\![\mathsf{while} : (\mathsf{exp}_1 \times \mathsf{com}_2) \longrightarrow \mathsf{com}_3]\!]$



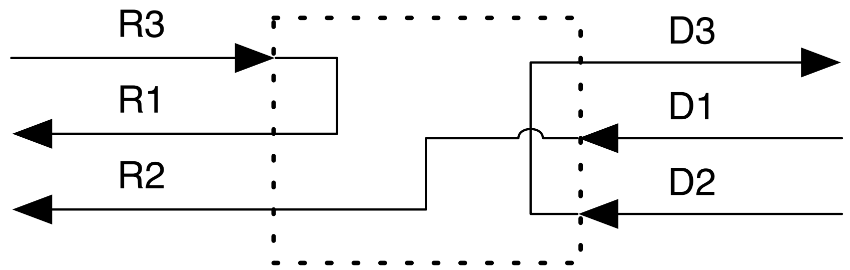
synchronous traces for constants

$[\![\mathsf{skip} : \mathsf{com}]\!]$

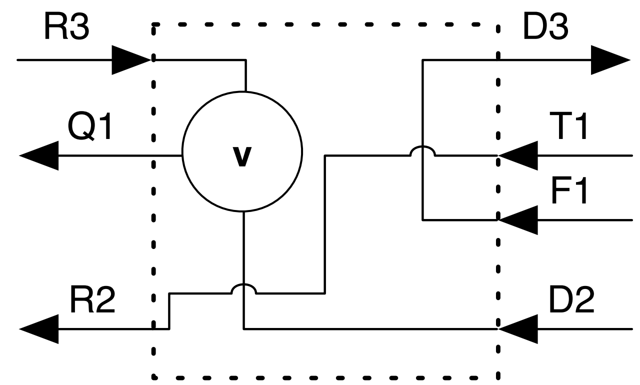


<R,D>

$[\![\mathsf{seq} : (\mathsf{com}_1 \times \mathsf{com}_2) \longrightarrow \mathsf{com}_3]\!]$



$[\![\mathsf{while} : (\mathsf{exp}_1 \times \mathsf{com}_2) \longrightarrow \mathsf{com}_3]\!]$

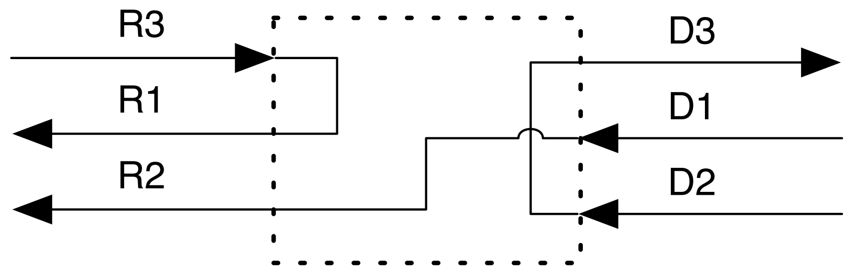


synchronous traces for constants

$[\![\mathsf{skip} : \mathsf{com}]\!]$

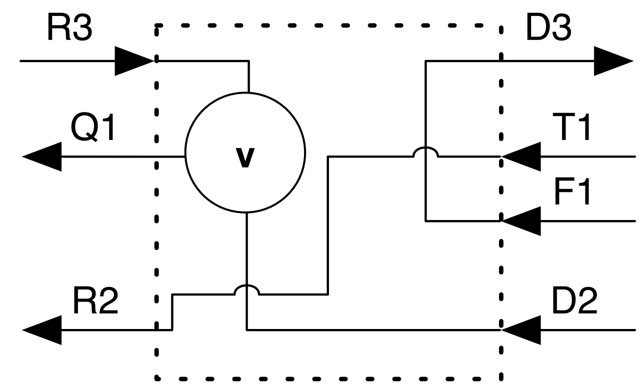
<R,D>

$[\![\mathsf{seq} : (\mathsf{com}_1 \times \mathsf{com}_2) \rightarrow \mathsf{com}_3]\!]$



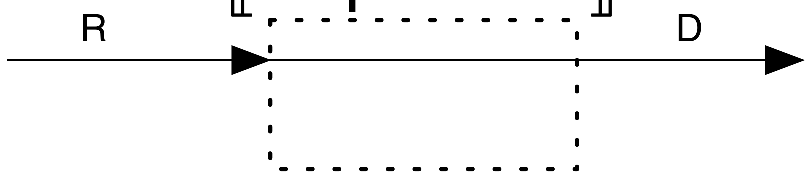
<R3,R1>.<D1,R2>.<D2,D3>

$[\![\mathsf{while} : (\mathsf{exp}_1 \times \mathsf{com}_2) \rightarrow \mathsf{com}_3]\!]$

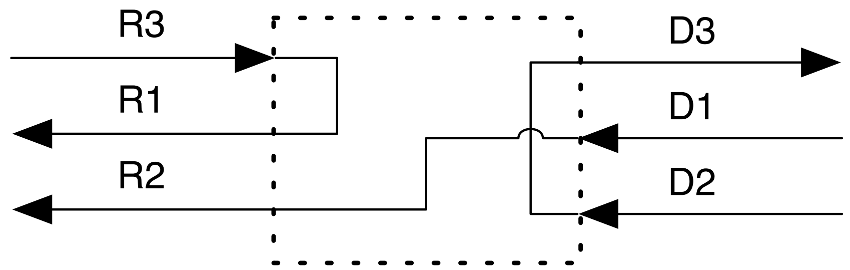


synchronous traces for constants

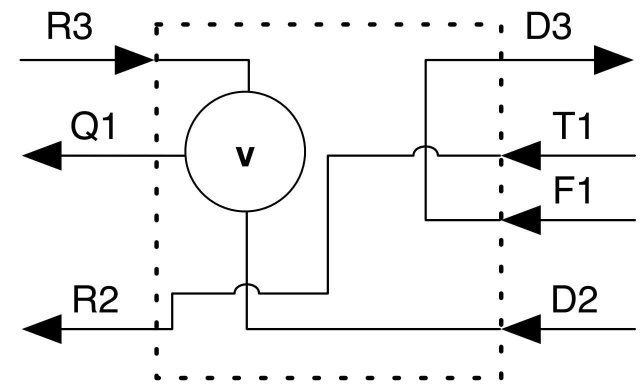
$[\![\text{skip} : \text{com}]\!]$



$[\![\text{seq} : (\text{com}_1 \times \text{com}_2) \rightarrow \text{com}_3]\!]$



$[\![\text{while} : (\text{exp}_1 \times \text{com}_2) \rightarrow \text{com}_3]\!]$



<R,D>

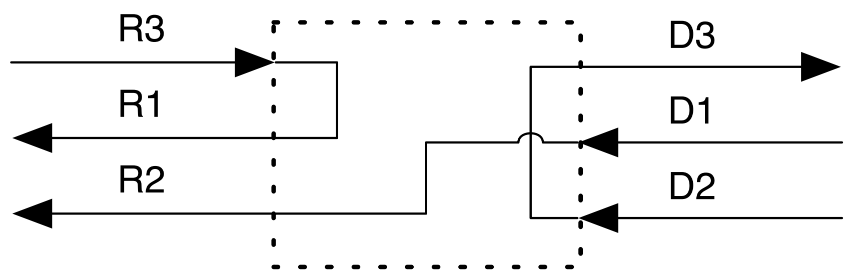
<R3,R1>.<D1,R2>.<D2,D3>

<R3,R1,D1,R2>.<D2,D3>

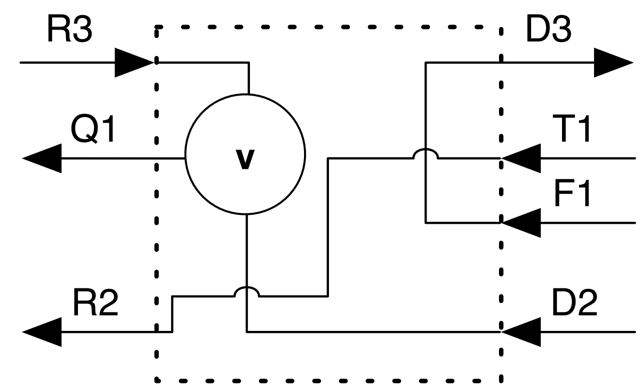
synchronous traces for constants

$[\![\mathsf{skip} : \mathsf{com}]\!]$

$[\![\mathsf{seq} : (\mathsf{com}_1 \times \mathsf{com}_2) \to \mathsf{com}_3]\!]$



$[\![\mathsf{while} : (\mathsf{exp}_1 \times \mathsf{com}_2) \to \mathsf{com}_3]\!]$



<R,D>

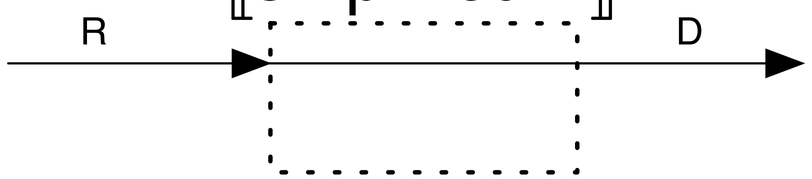
<R3,R1>.<D1,R2>.<D2,D3>

<R3,R1,D1,R2>.<D2,D3>

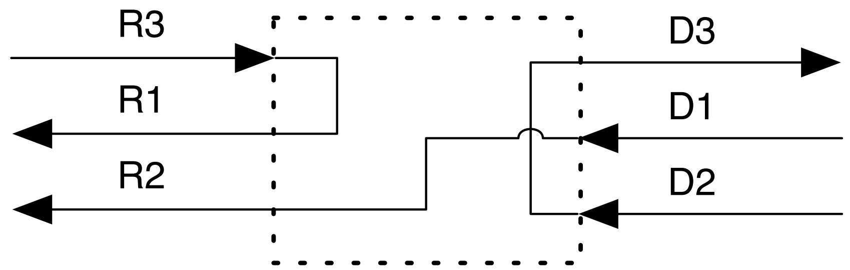
<R3,R1>.<D1,R2,D2,D3>

synchronous traces for constants

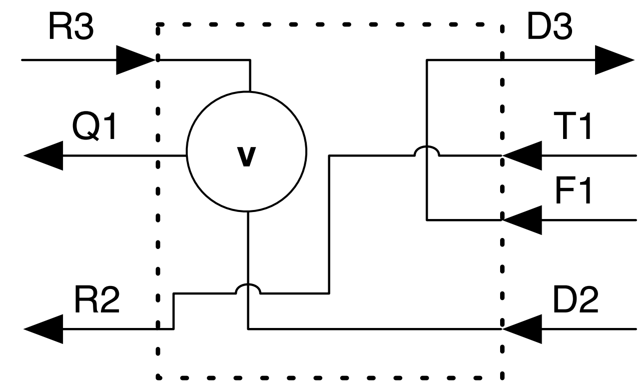
$[\![\mathsf{skip} : \mathsf{com}]\!]$



$[\![\mathsf{seq} : (\mathsf{com}_1 \times \mathsf{com}_2) \to \mathsf{com}_3]\!]$



$[\![\mathsf{while} : (\mathsf{exp}_1 \times \mathsf{com}_2) \to \mathsf{com}_3]\!]$



<R,D>

<R3,R1>.<D1,R2>.<D2,D3>

<R3,R1,D1,R2>.<D2,D3>

<R3,R1>.<D1,R2,D2,D3>

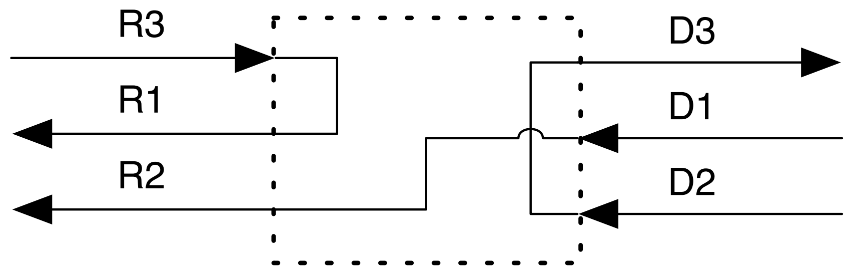
<R3,R1,D1,R2,D2,D3>

synchronous traces for constants

$[\![\mathsf{skip} : \mathsf{com}]\!]$

<R,D>

$[\![\mathsf{seq} : (\mathsf{com}_1 \times \mathsf{com}_2) \to \mathsf{com}_3]\!]$



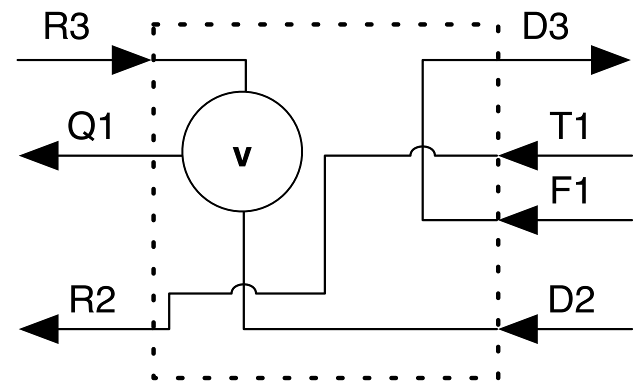
<R3,R1>.<D1,R2>.<D2,D3>

<R3,R1,D1,R2>.<D2,D3>

<R3,R1>.<D1,R2,D2,D3>

<R3,R1,D1,R2,D2,D3>

$[\![\mathsf{while} : (\mathsf{exp}_1 \times \mathsf{com}_2) \to \mathsf{com}_3]\!]$



<R3,Q1>.<T1,R2>

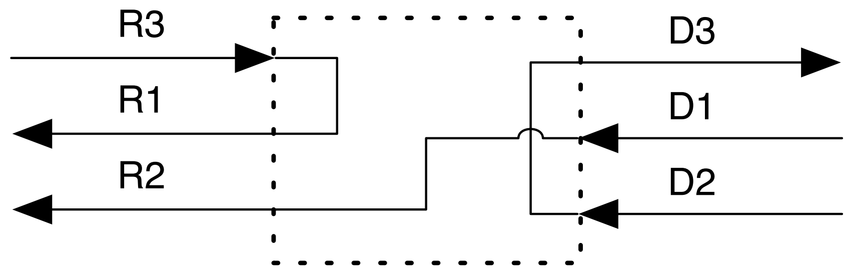
<D2,Q1>.<F1,D3>

synchronous traces for constants

$[\![\text{skip} : \text{com}]\!]$

<R,D>

$[\![\text{seq} : (\text{com}_1 \times \text{com}_2) \rightarrow \text{com}_3]\!]$



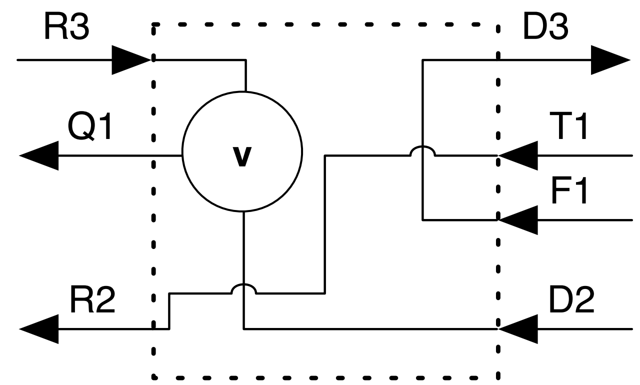
<R3,R1>.<D1,R2>.<D2,D3>

<R3,R1,D1,R2>.<D2,D3>

<R3,R1>.<D1,R2,D2,D3>

<R3,R1,D1,R2,D2,D3>

$[\![\text{while} : (\text{exp}_1 \times \text{com}_2) \rightarrow \text{com}_3]\!]$



<R3,Q1>.<T1,R2>
<D2,Q1>.<F1,D3>

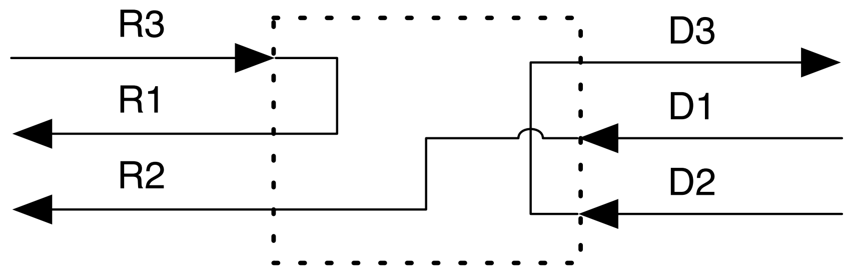
<R3,Q1,T1,R2>
<D2,Q1>.<F1,D3>

synchronous traces for constants

$[\![\mathsf{skip} : \mathsf{com}]\!]$

<R,D>

$[\![\mathsf{seq} : (\mathsf{com}_1 \times \mathsf{com}_2) \to \mathsf{com}_3]\!]$



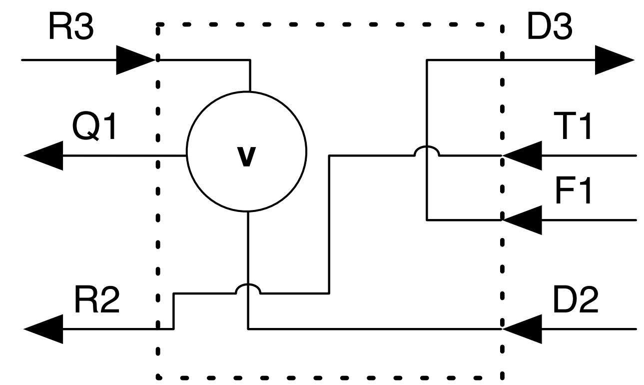
<R3,R1>.<D1,R2>.<D2,D3>

<R3,R1,D1,R2>.<D2,D3>

<R3,R1>.<D1,R2,D2,D3>

<R3,R1,D1,R2,D2,D3>

$[\![\mathsf{while} : (\mathsf{exp}_1 \times \mathsf{com}_2) \to \mathsf{com}_3]\!]$



<R3,Q1>.<T1,R2>
<D2,Q1>.<F1,D3>

<R3,Q1,T1,R2>
<D2,Q1>.<F1,D3>

<R3,Q1,T1,R2,D2,Q1>
<F1,D3>

more complex languages: *scc*

$$\frac{\Gamma, x : \theta^{m}, y : \theta^{n} \vdash_r M : \theta'}{\Gamma, x : \theta^{m+n} \vdash_r M[x/y] : \theta'}$$

more complex languages: *scc*

$$\frac{\Gamma, x : \theta^{m}, y : \theta^{n} \vdash_r M : \theta'}{\Gamma, x : \theta^{m+n} \vdash_r M[x/y] : \theta'}$$

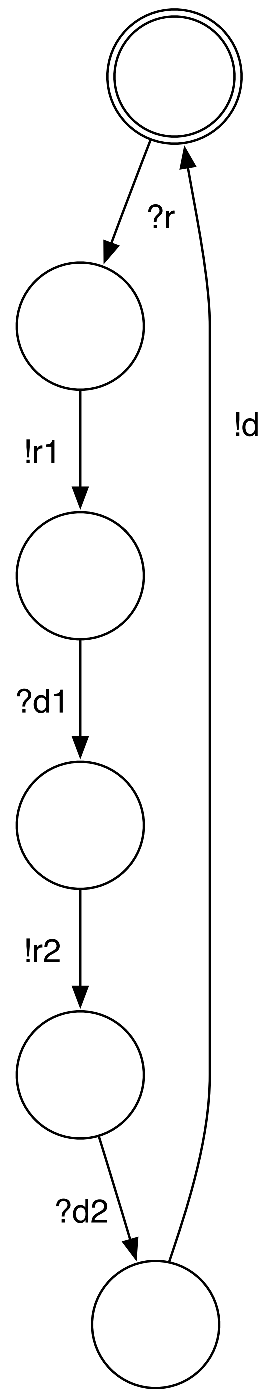
$$A^n \longrightarrow B \equiv \underbrace{!_{\odot}A \otimes \cdots \otimes !_{\odot}A}_{n} \multimap B$$

$$[\![A \odot B]\!] = [\![A]\!][\![B]\!] \cup [\![B]\!][\![A]\!] \qquad [\![!_{\odot}A]\!] = [\![A]\!]^*$$

why not represent the game models in hardware?

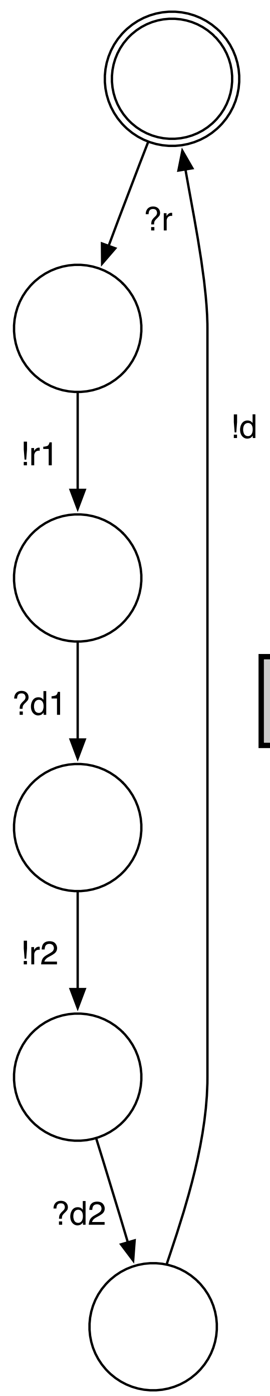
why not represent the game models in hardware?

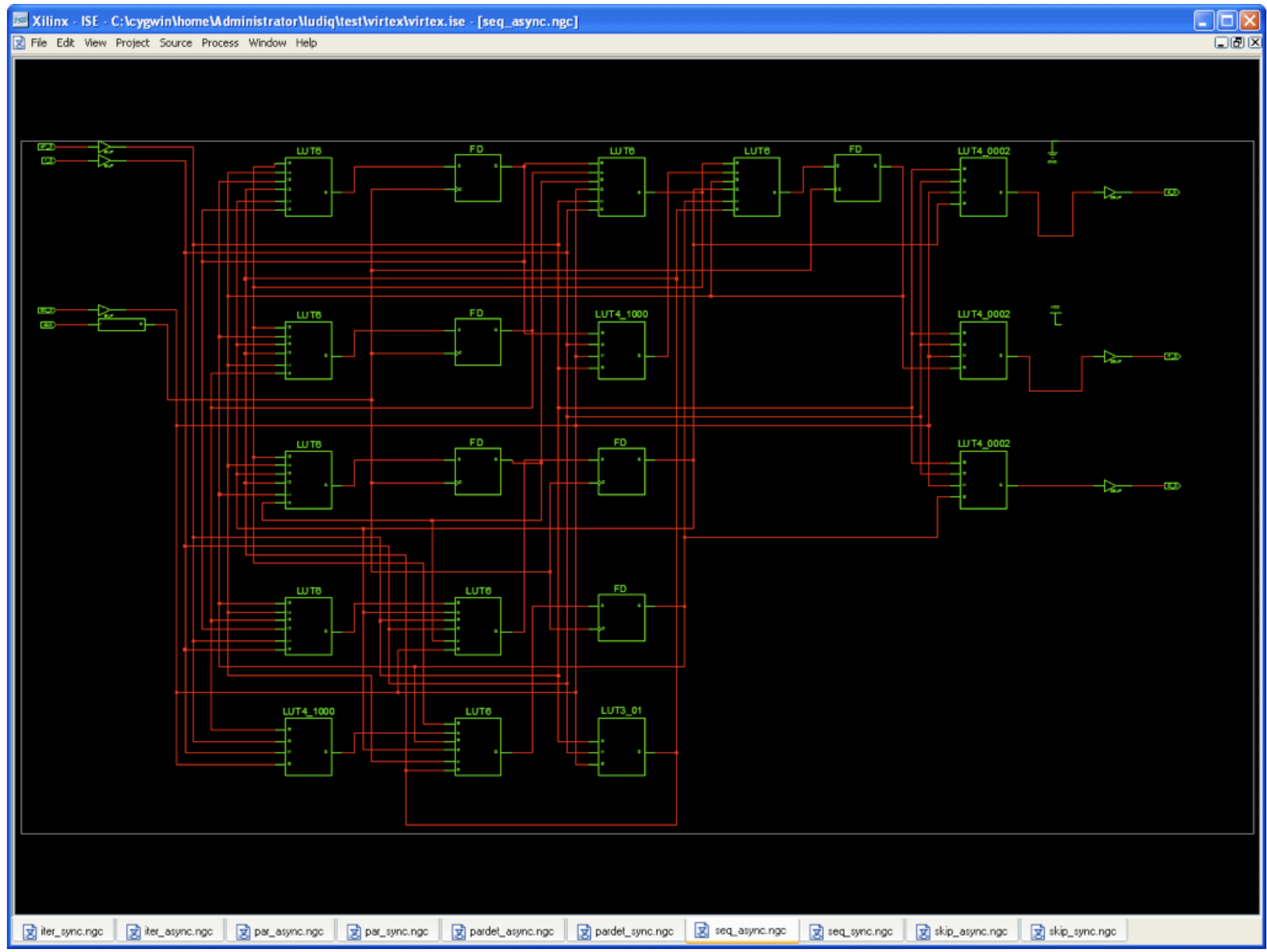
$$\texttt{seq} : \texttt{com} \times \texttt{com} \Rightarrow \texttt{com}$$



why not represent the game models in hardware?

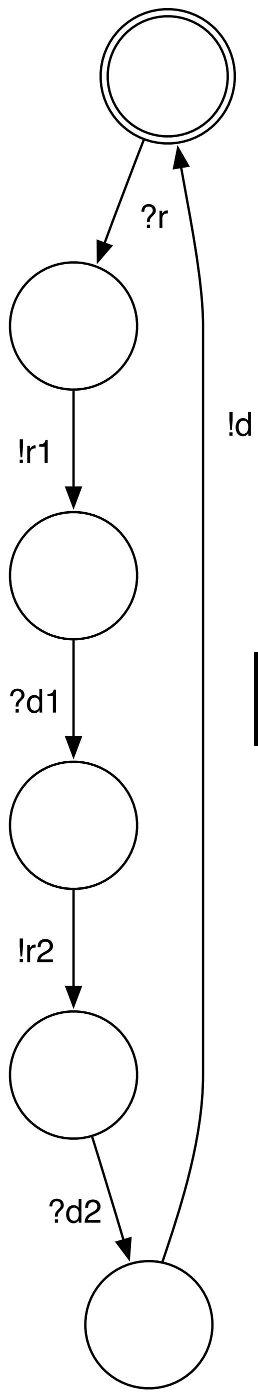
$$\texttt{seq} : \texttt{com} \times \texttt{com} \Rightarrow \texttt{com}$$

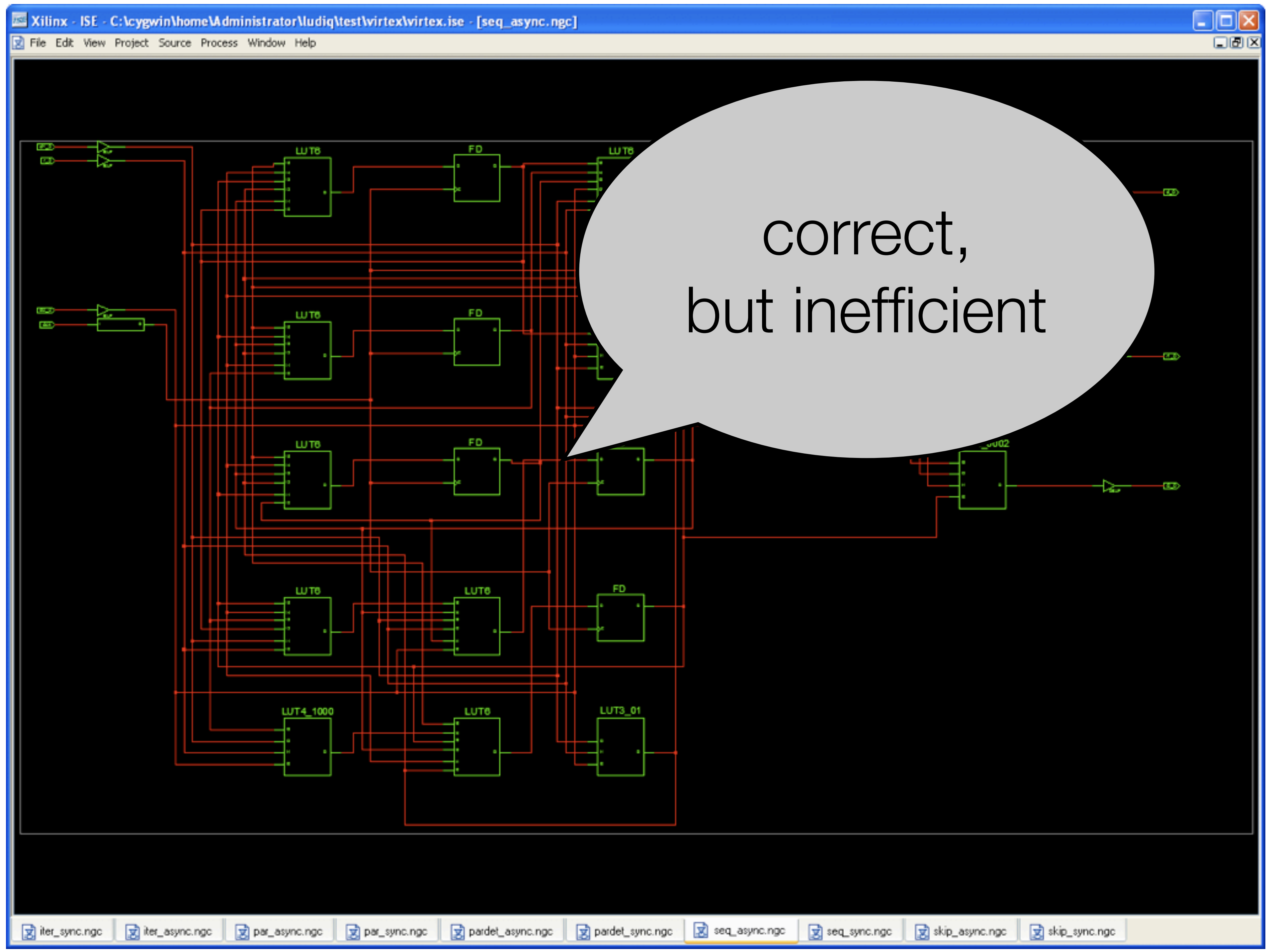




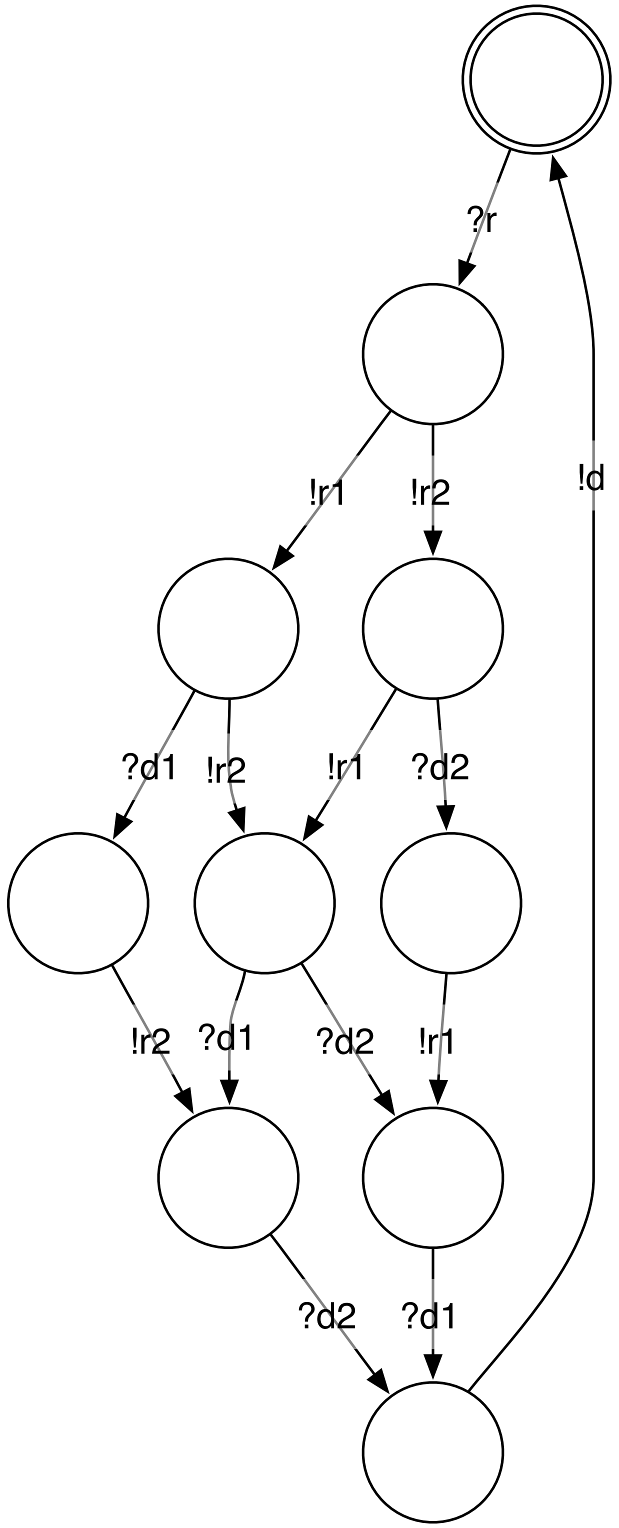
why not represent the game models in hardware?

$$\texttt{seq} : \texttt{com} \times \texttt{com} \Rightarrow \texttt{com}$$

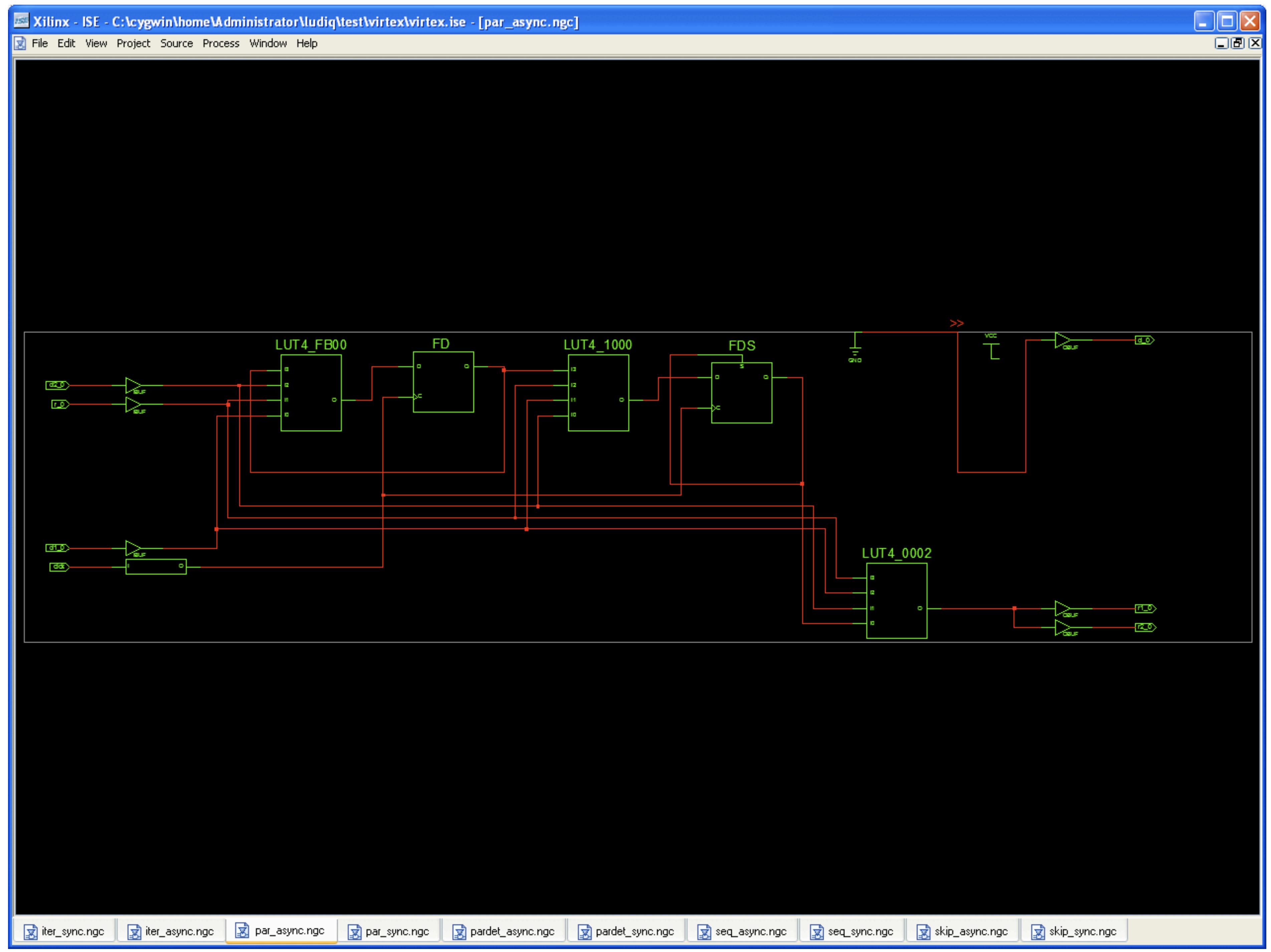




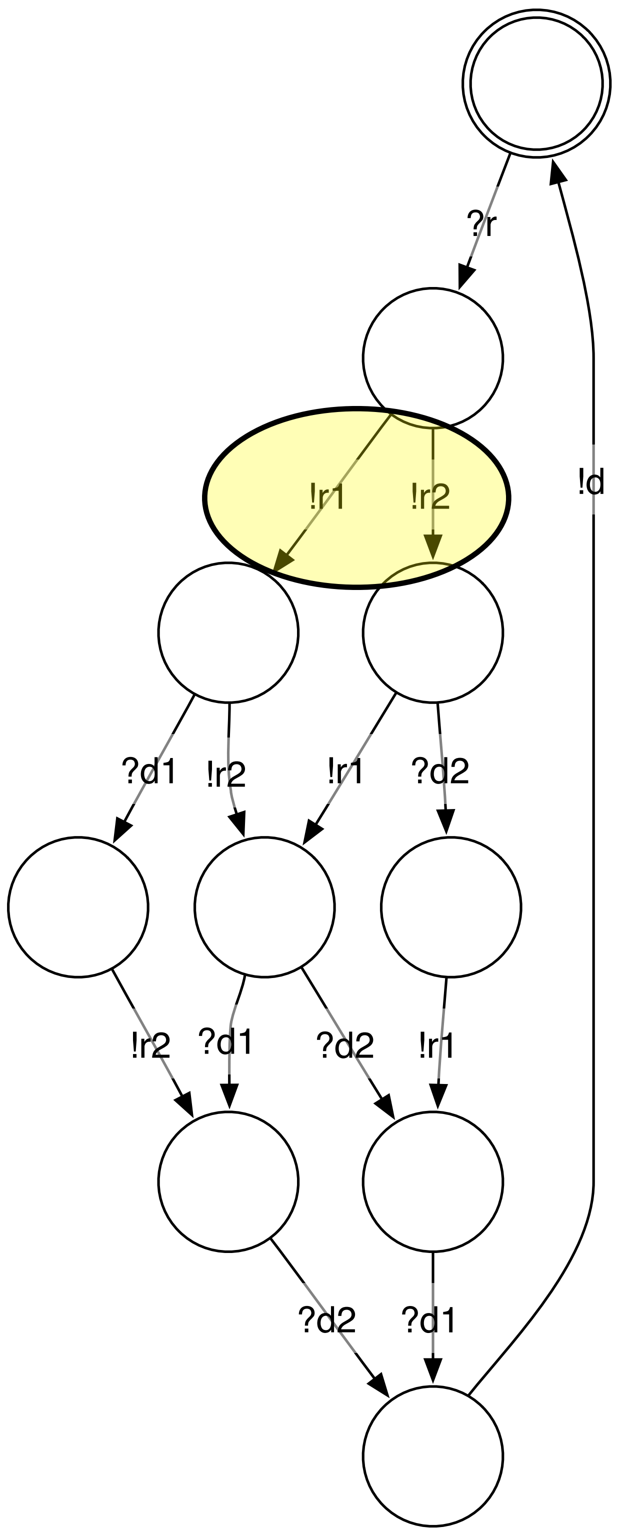
more on representing game models in hardware



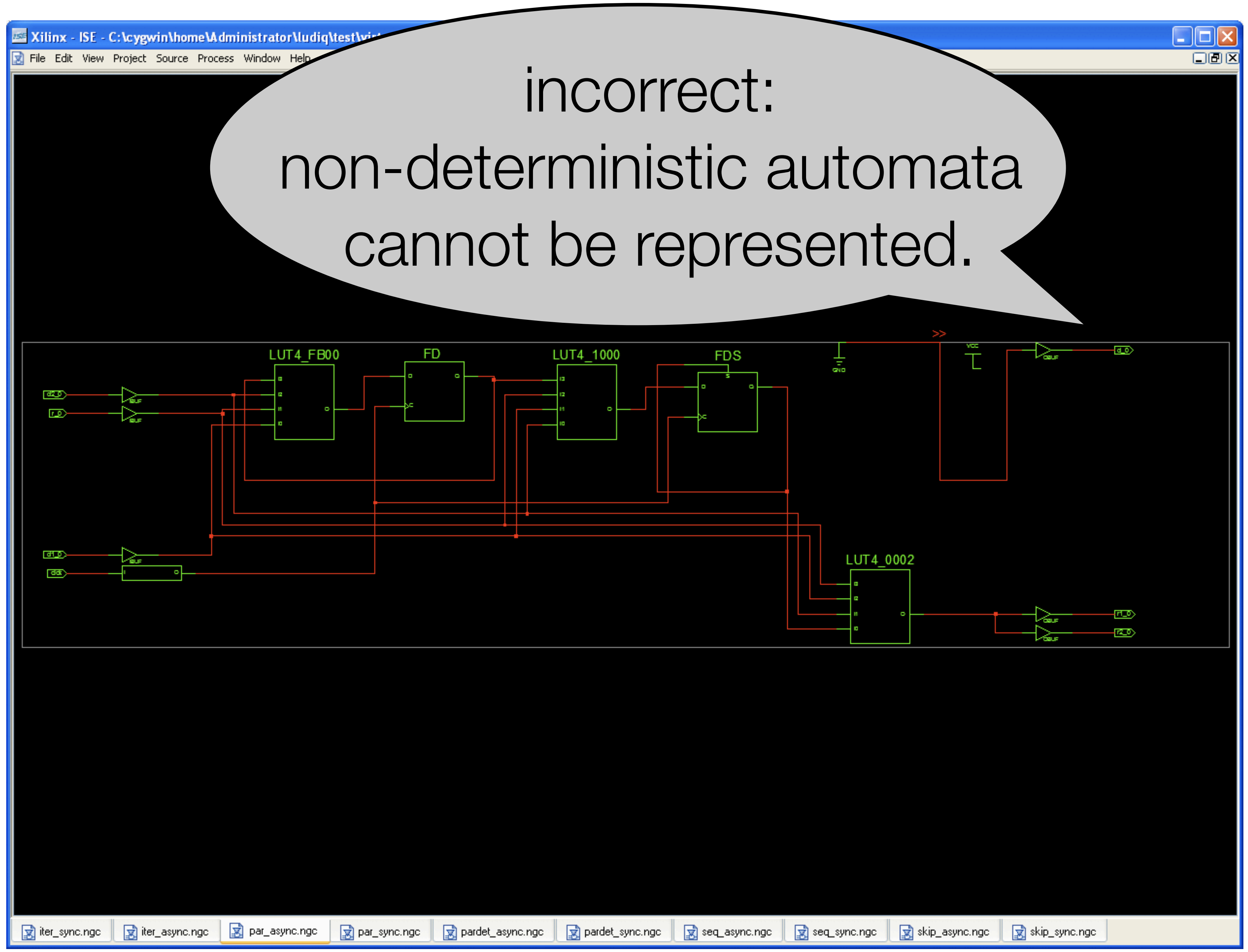

```
par : com -> com -> com
```



more on representing game models in hardware




```
par : com -> com -> com
```



incorrect:
non-deterministic automata
cannot be represented.

a solution: round abstraction

a solution: round abstraction

- *“Reactive Modules”,* Alur & Henzinger. LICS 1996 / FMSD 1999.

a solution: round abstraction

- *“Reactive Modules”,* Alur & Henzinger. LICS 1996 / FMSD 1999.
- create synchronous “rounds” of signals controlled by specific signals used as “clocks”

a solution: round abstraction

- *“Reactive Modules”,* Alur & Henzinger. LICS 1996 / FMSD 1999.
- create synchronous “rounds” of signals controlled by specific signals used as “clocks”
- we use a “maximal” form of round abstraction

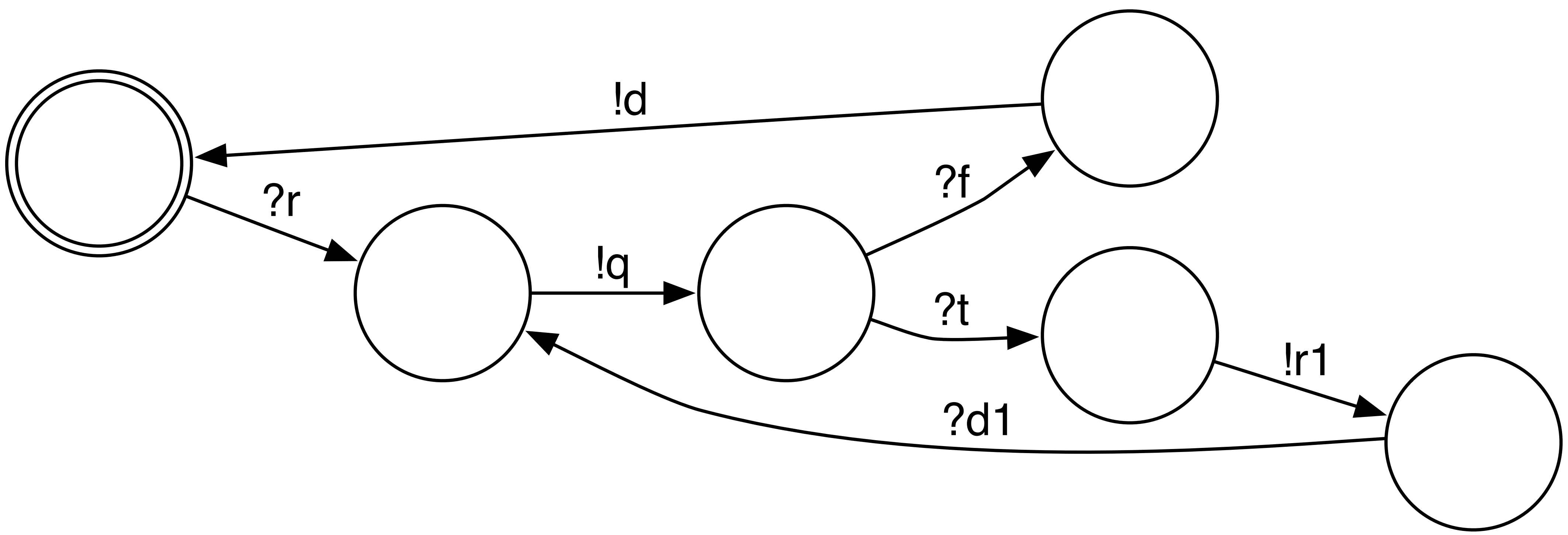
a solution: round abstraction

- *“Reactive Modules”,* Alur & Henzinger. LICS 1996 / FMSD 1999.
- create synchronous “rounds” of signals controlled by specific signals used as “clocks”
- we use a “maximal” form of round abstraction
    - make the rounds as long as possible

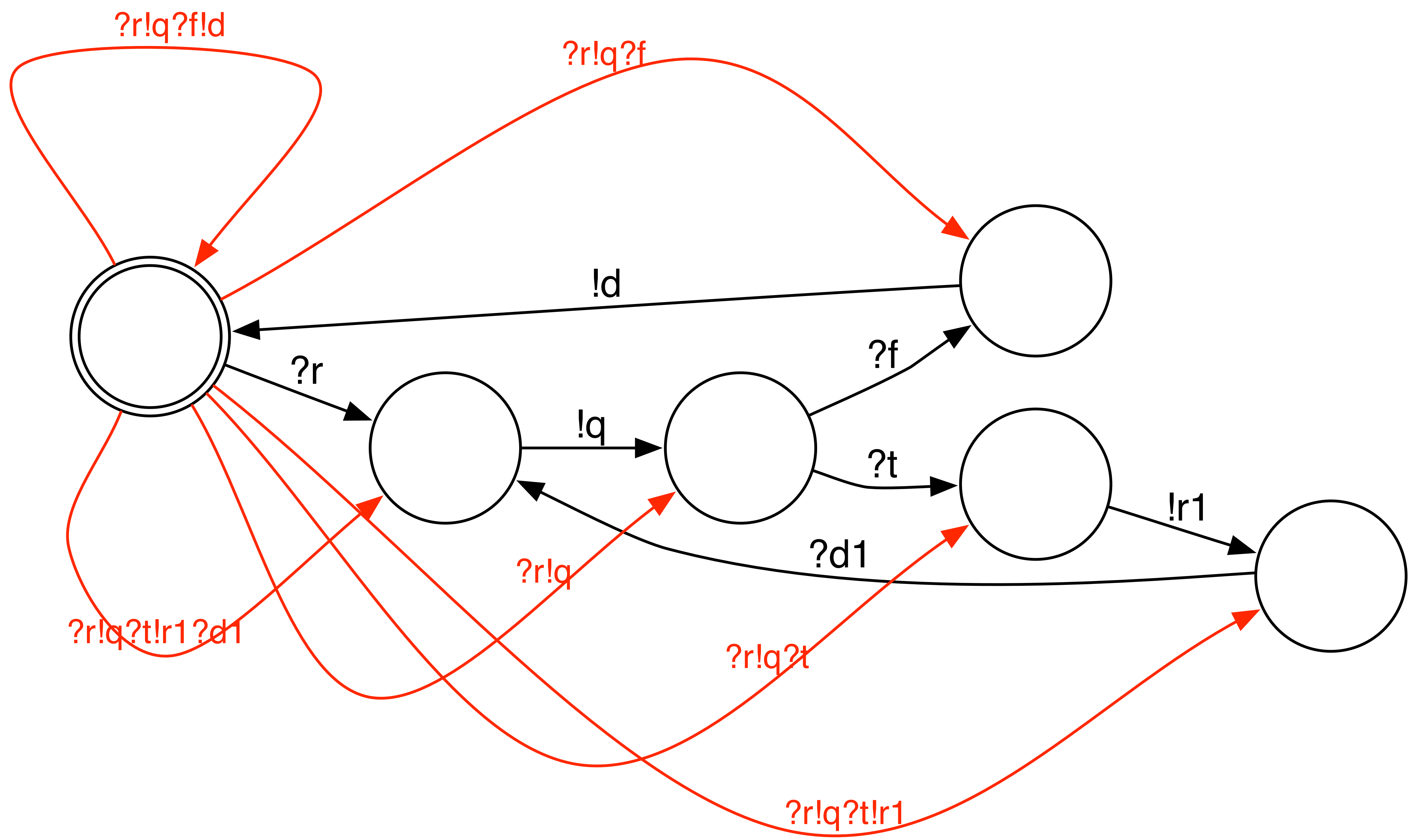
a solution: round abstraction

- *"Reactive Modules",* Alur & Henzinger. LICS 1996 / FMSD 1999.
- create synchronous "rounds" of signals controlled by specific signals used as "clocks"
- we use a "maximal" form of round abstraction
    - make the rounds as long as possible
    - ... but avoid using the same signal twice in one round (cf "schizophrenia" in Esterel)

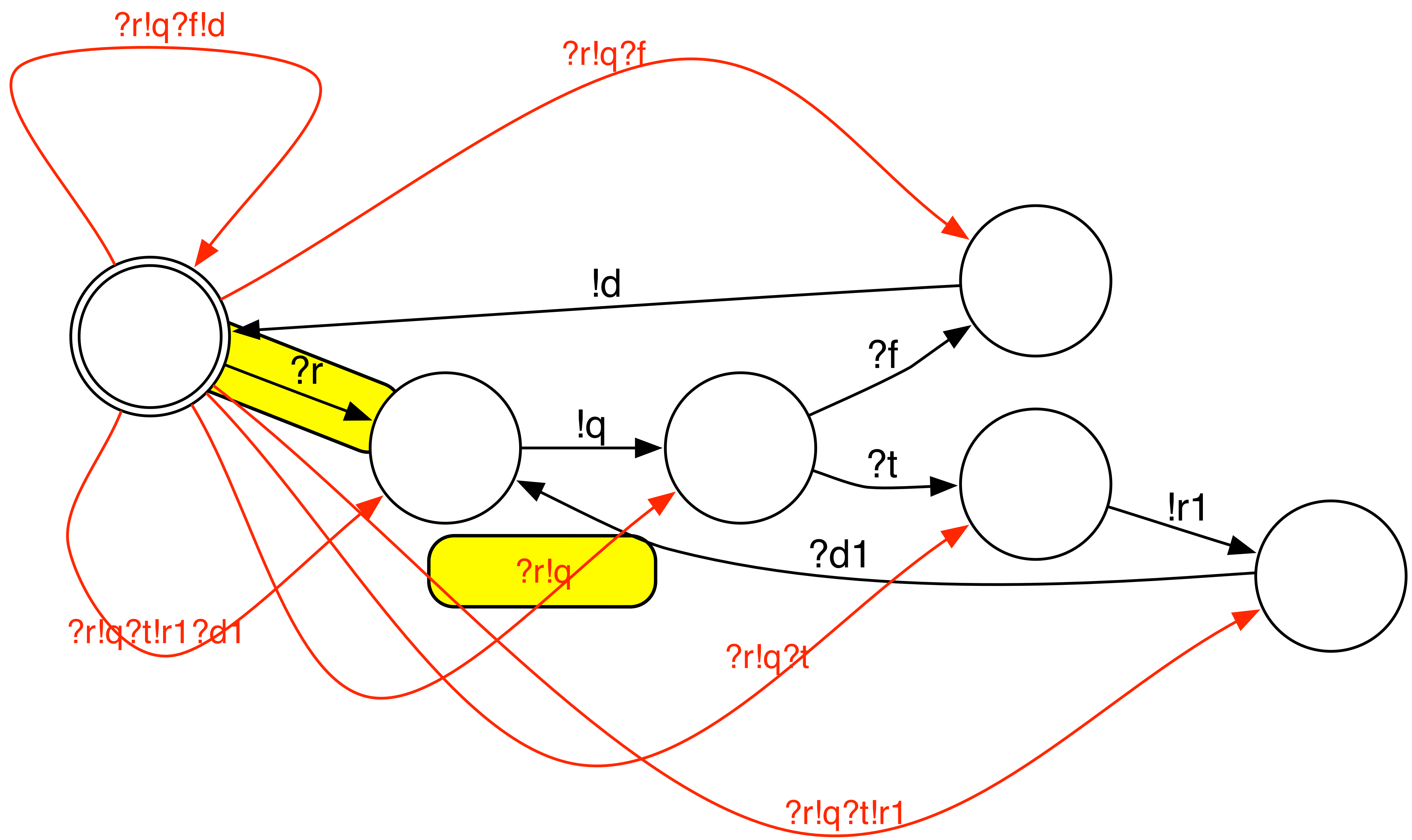
asynchronous automaton: `while`



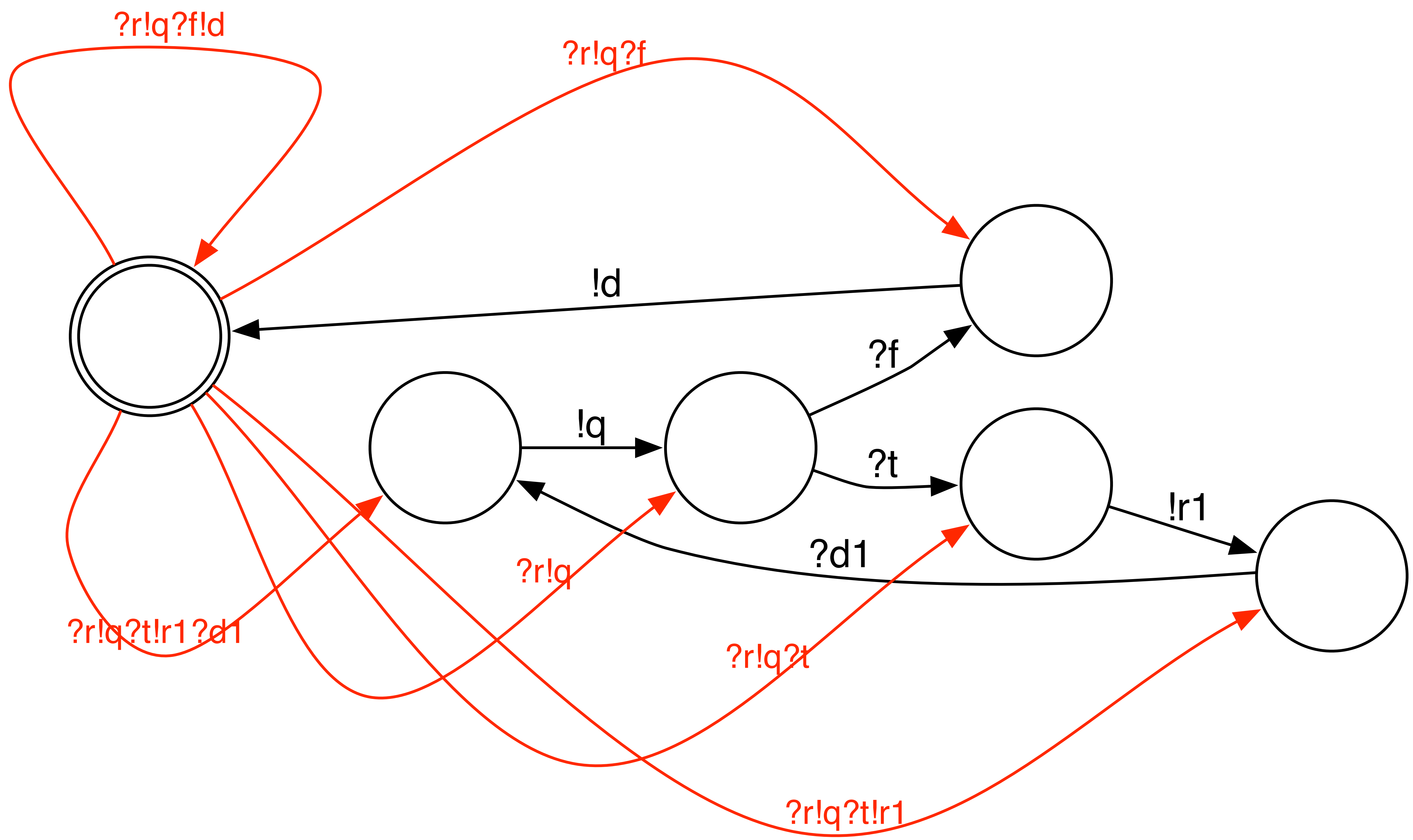
step 1: round generation



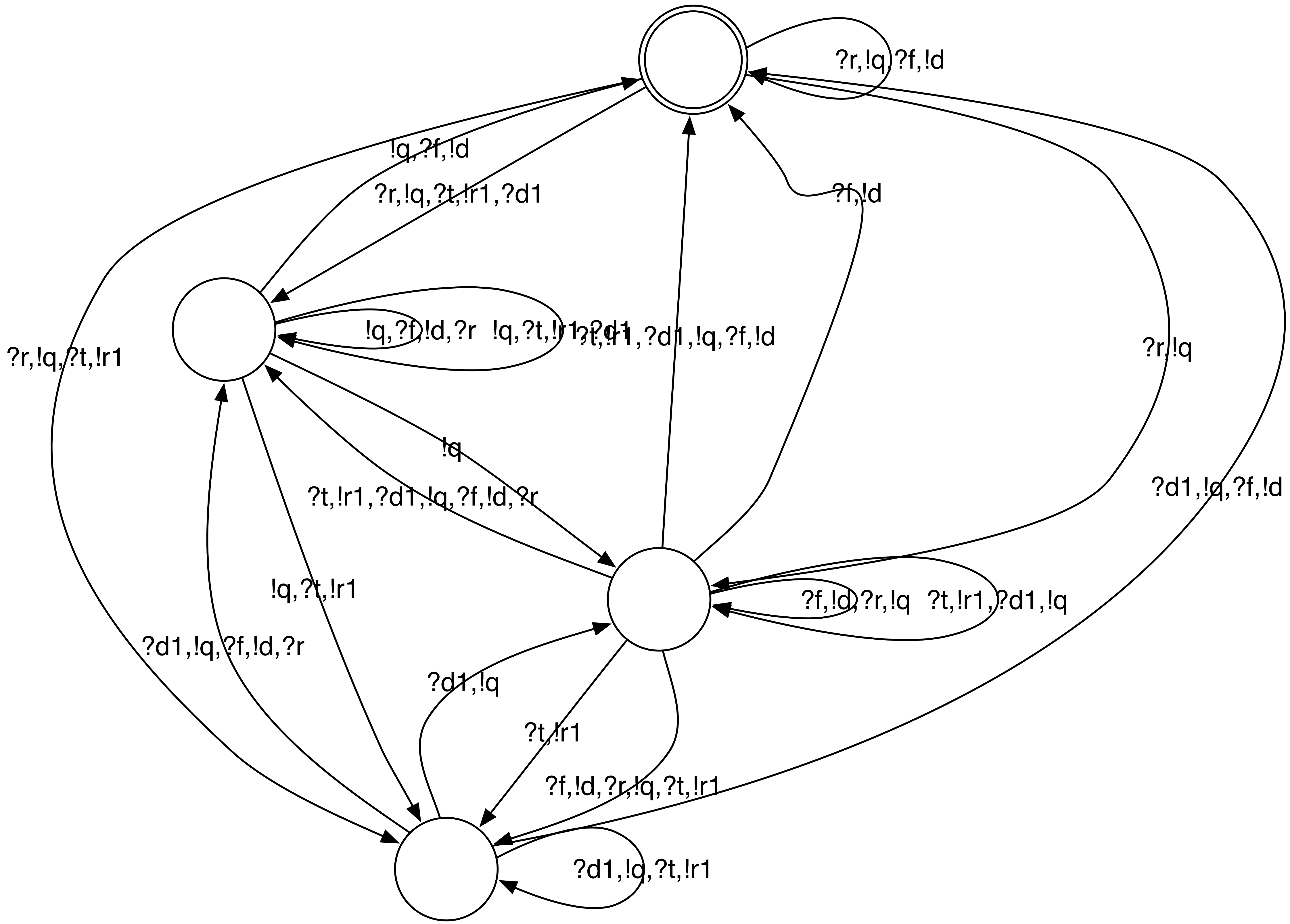
step 1: round generation

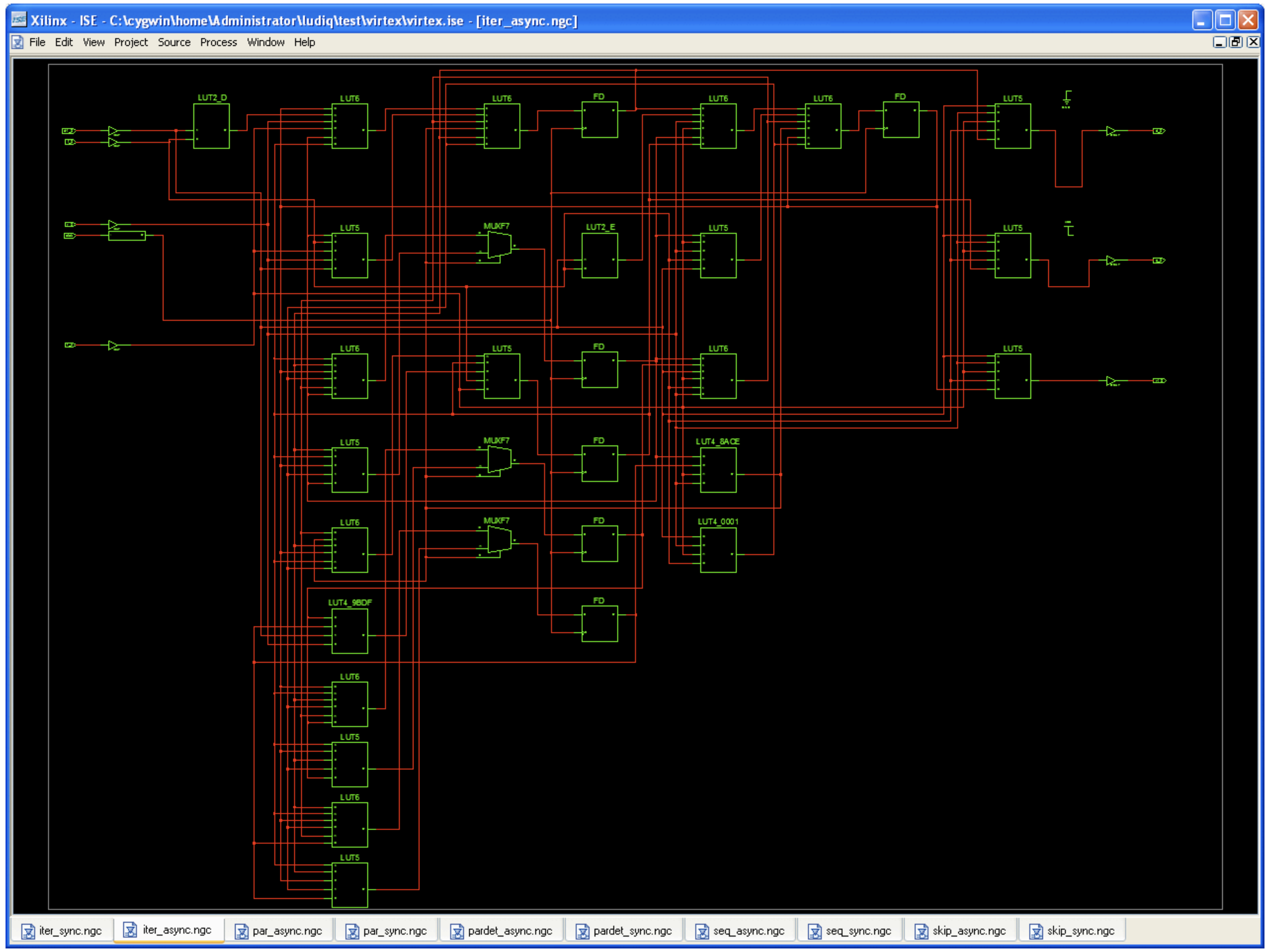


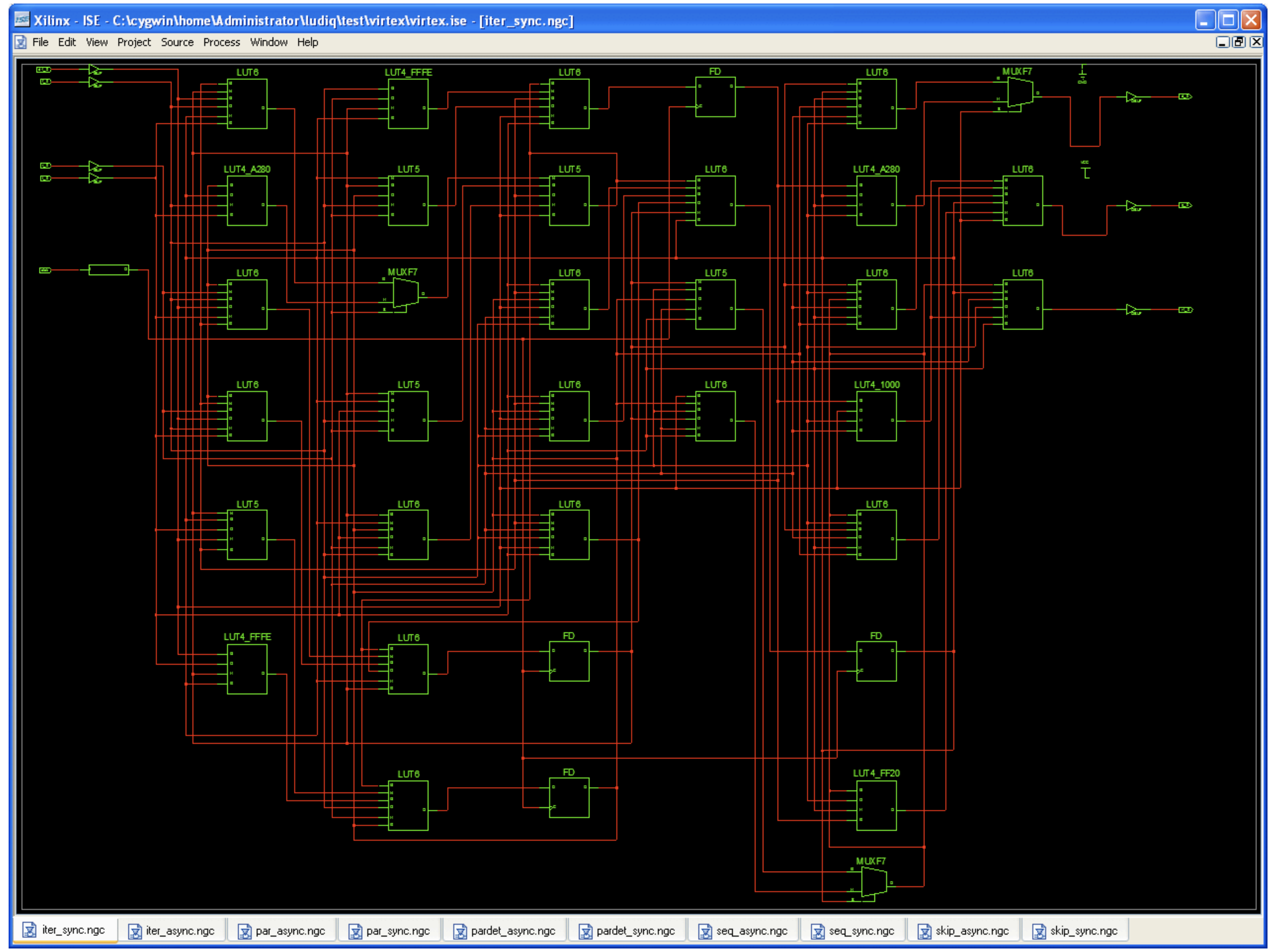
Step 2: reduction



synchronous automaton for `while`



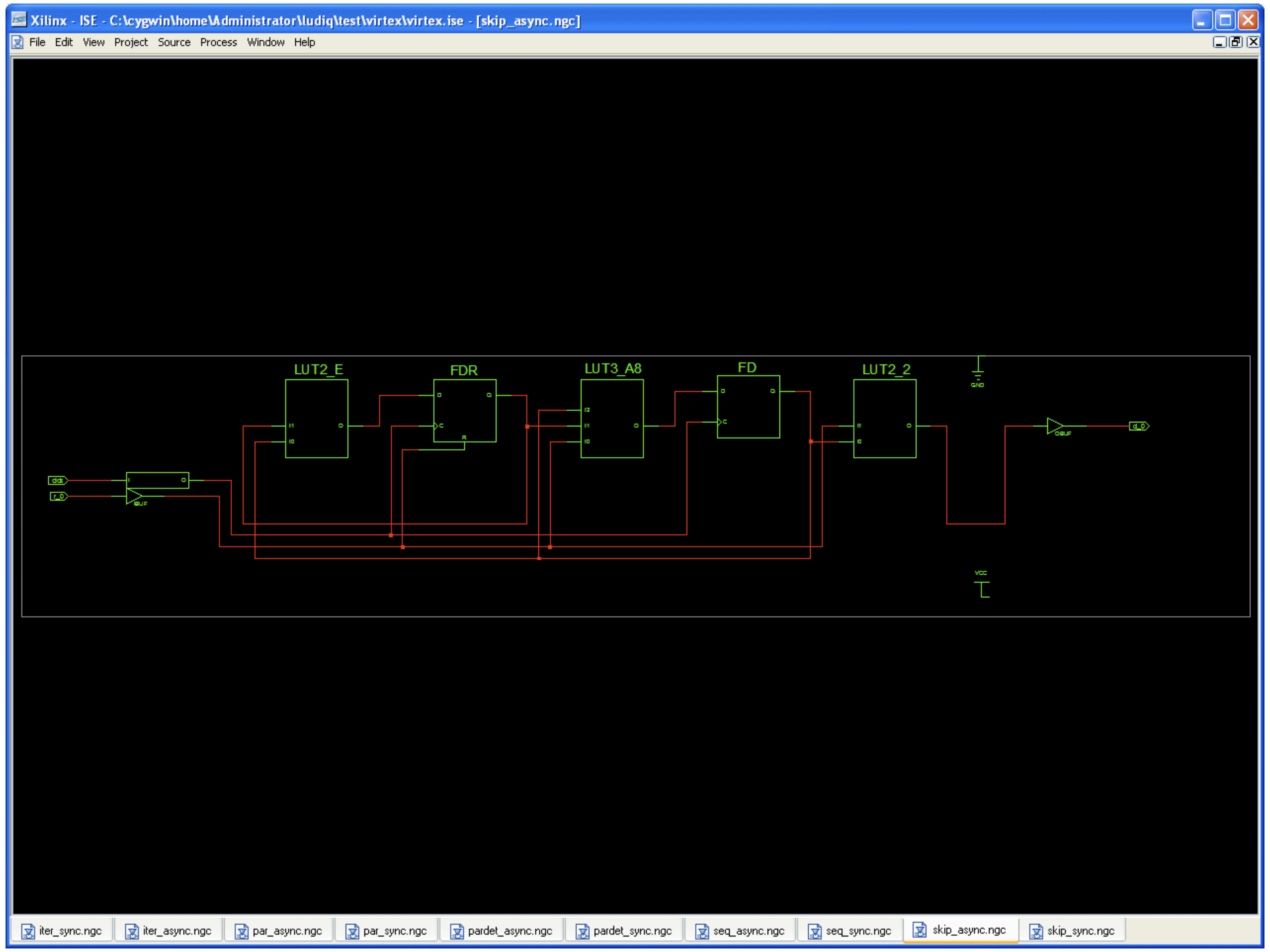


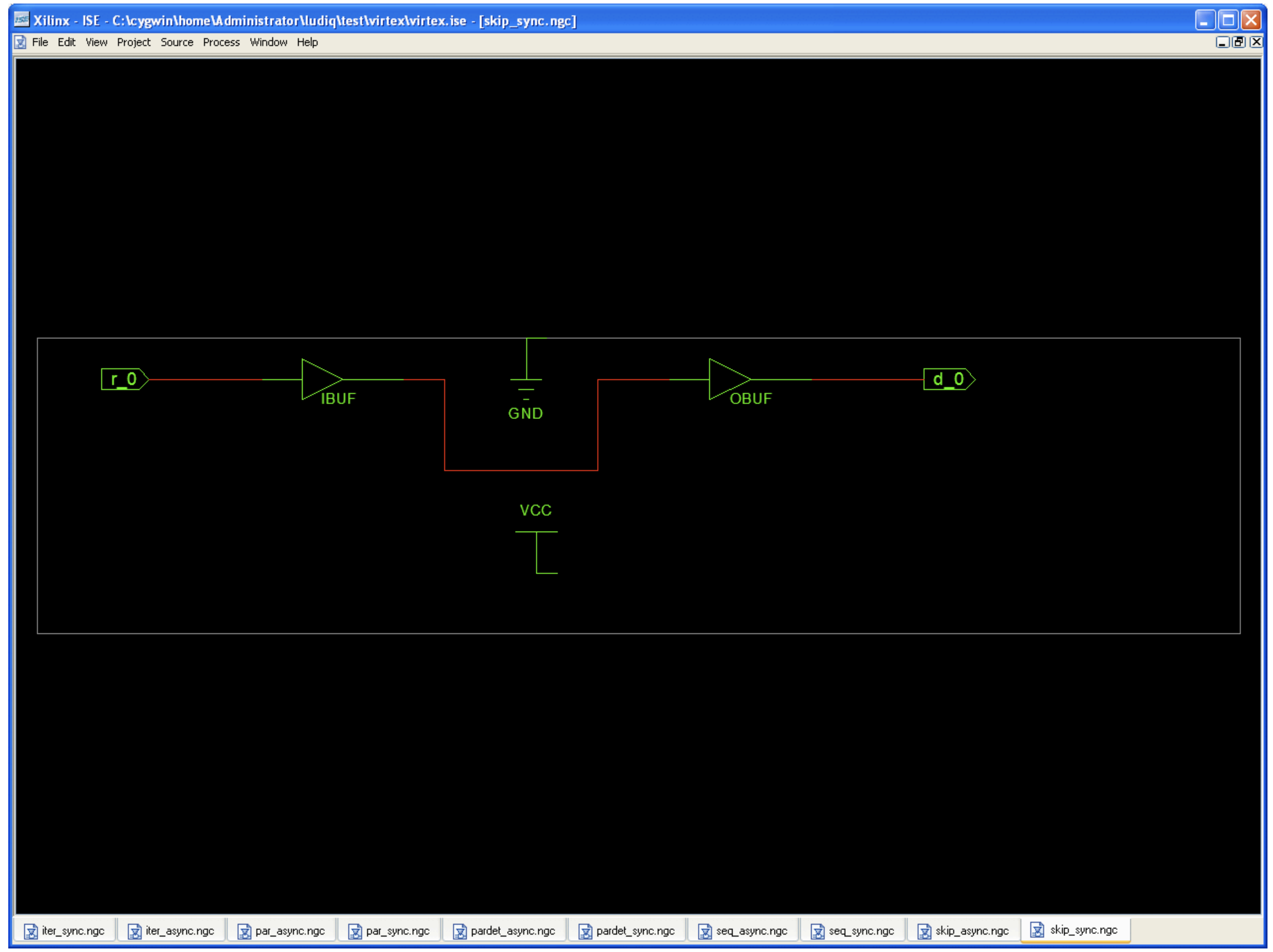


6 states, 23 LUTs

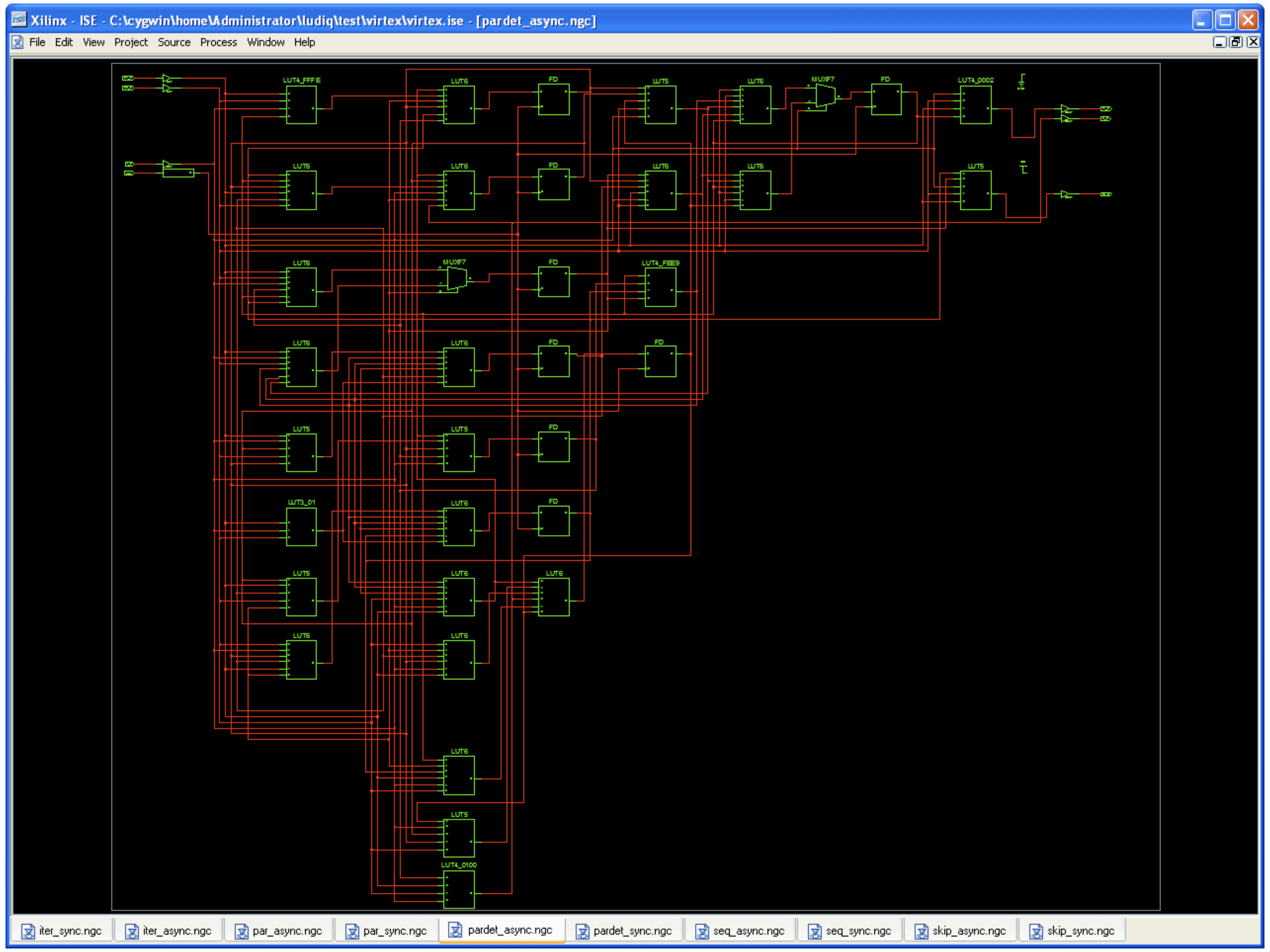
4 states, 28 LUTs

asynchronous versus synchronous representations for iteration

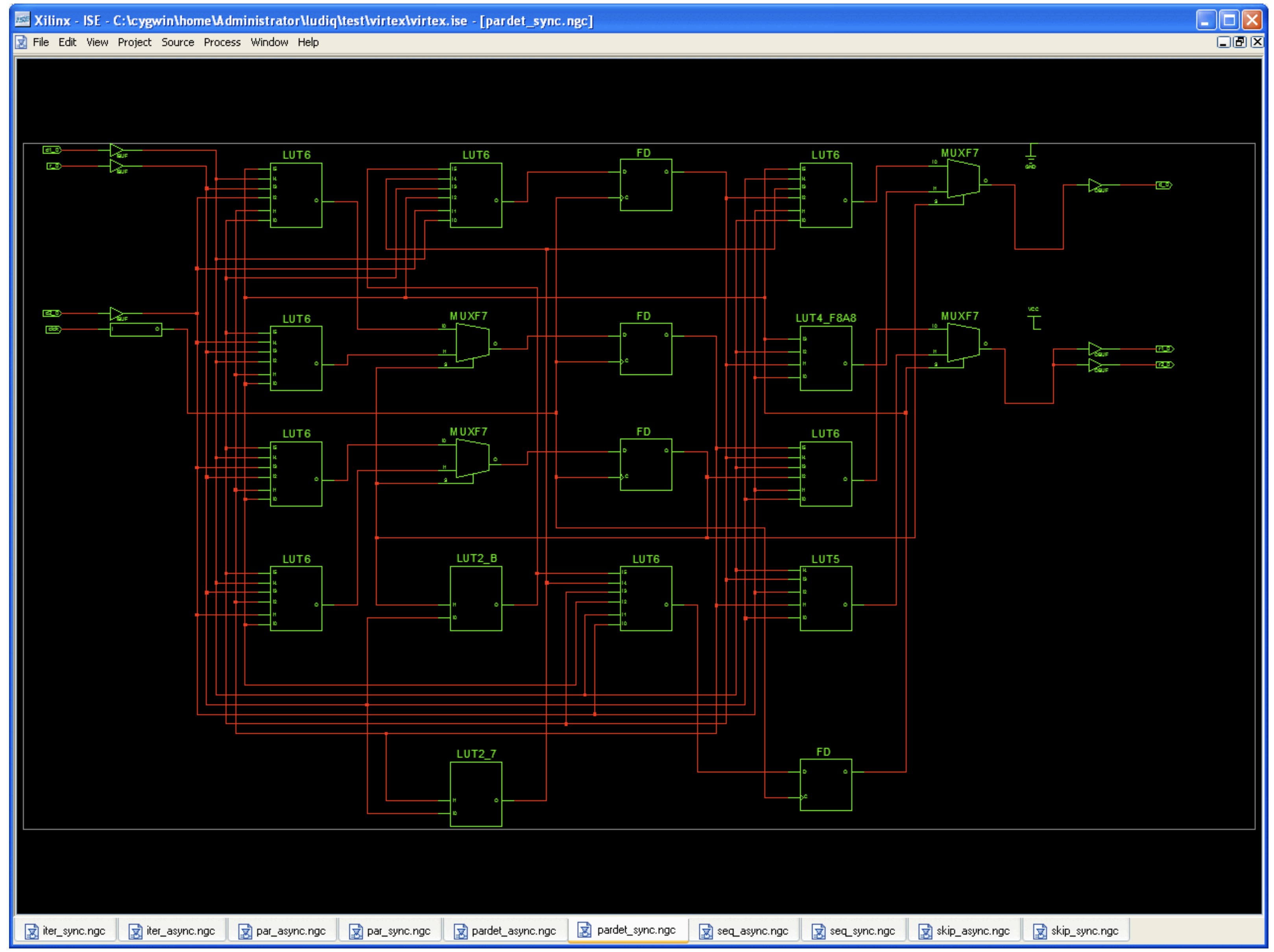




asynchronous versus synchronous representations for skip

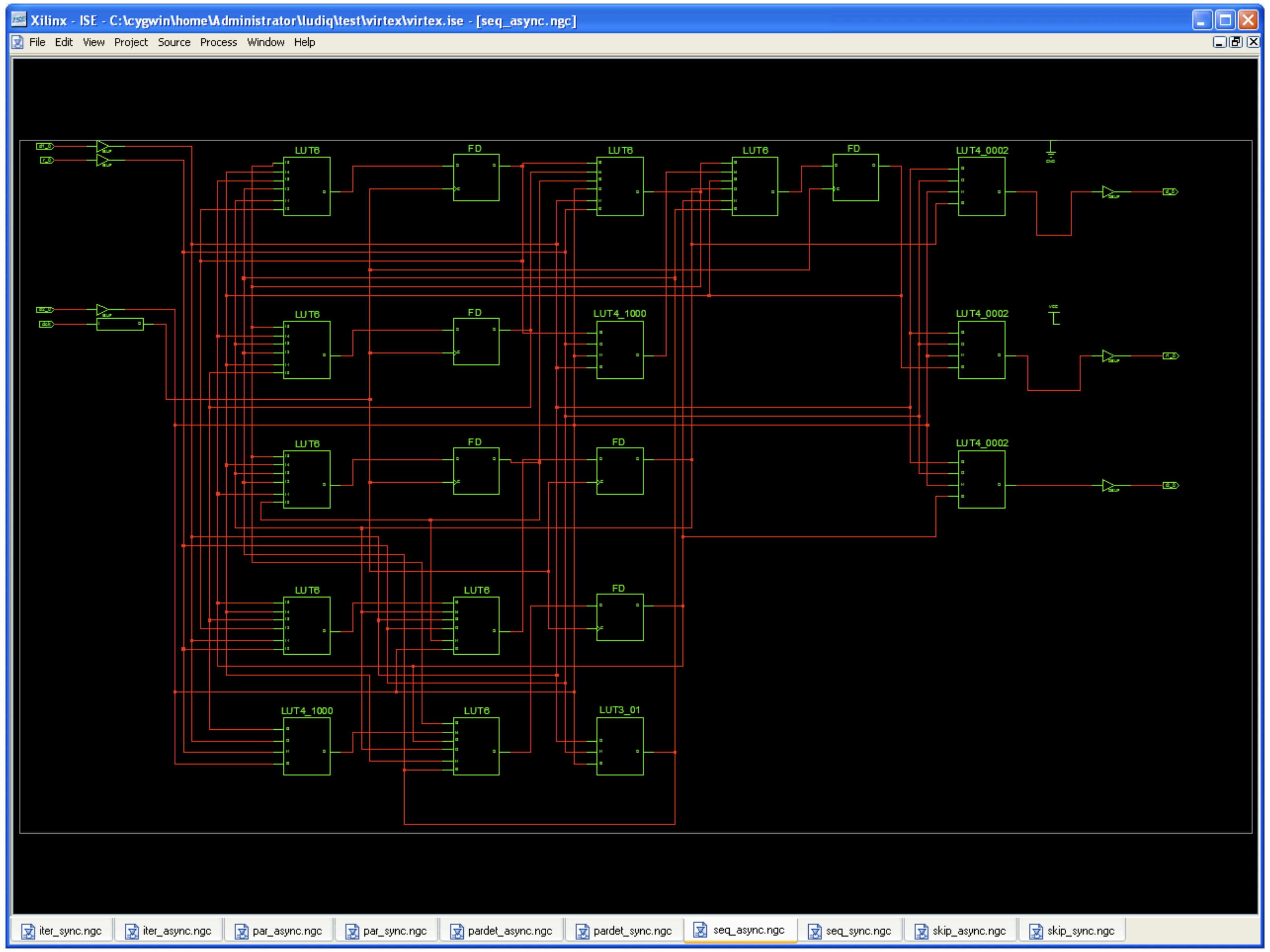


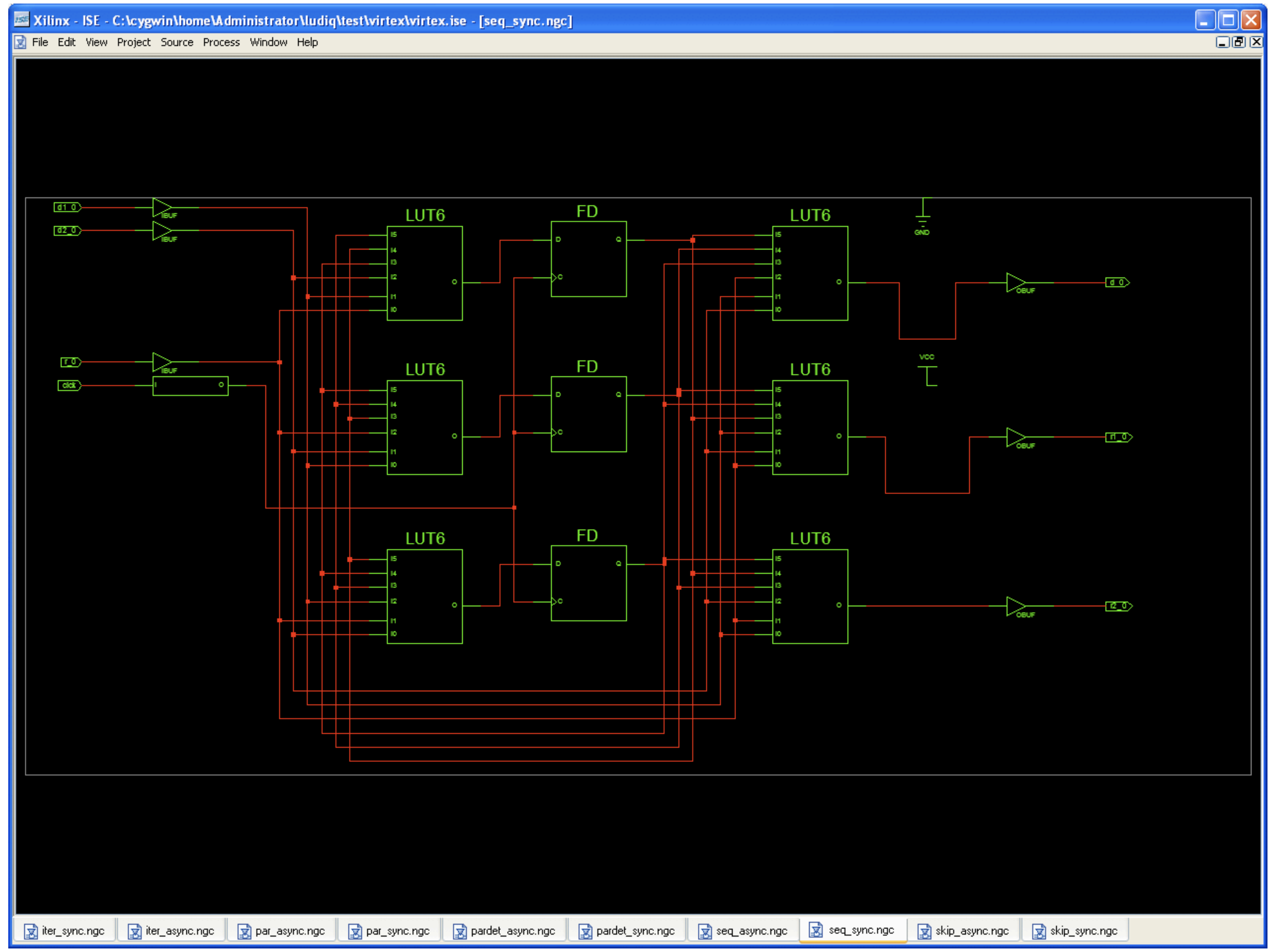
8 states, 26 LUTs



4 states, 12 LUTs

async vs sync representations for (deterministic) parallel composition

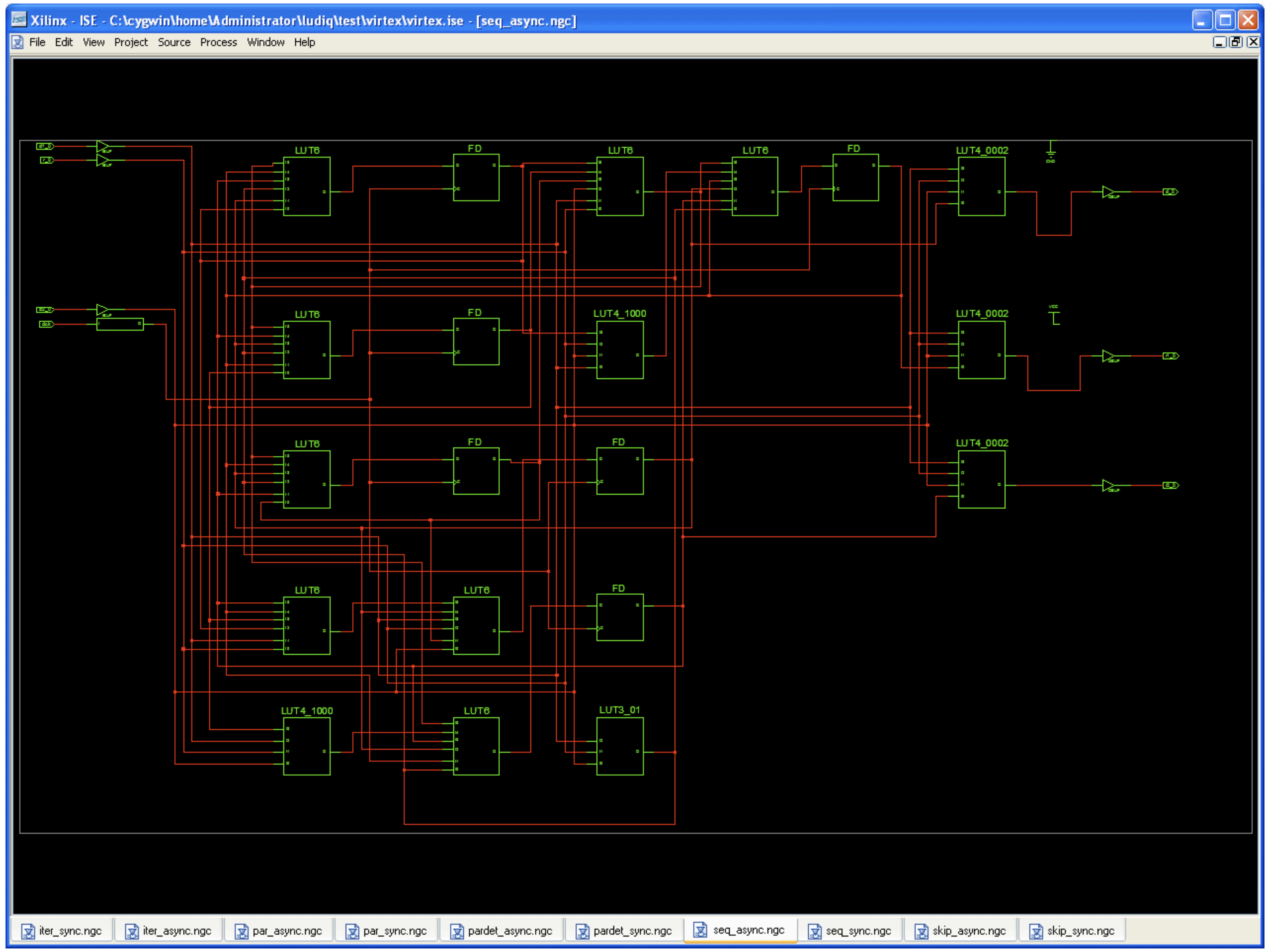




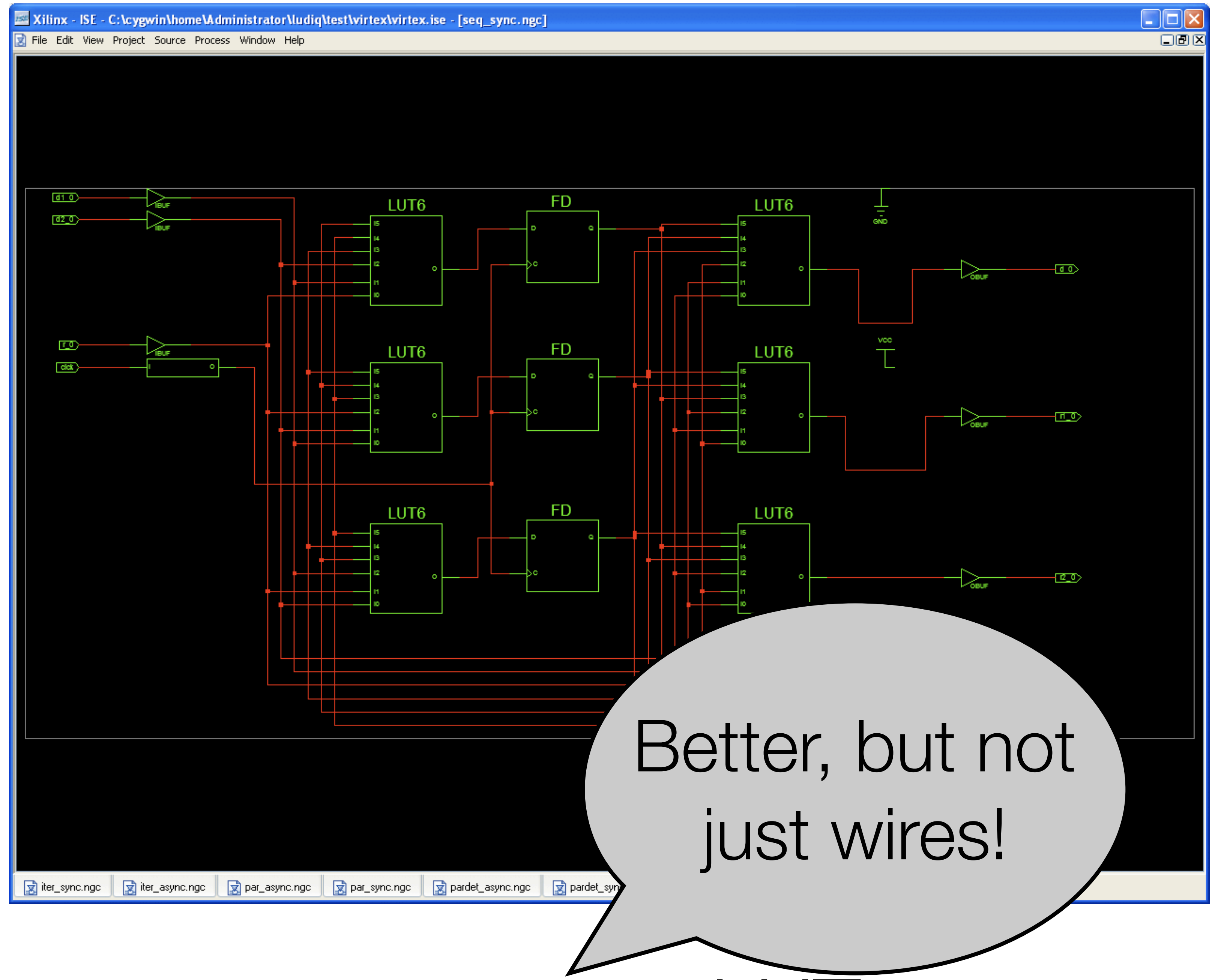
8 states, 26 LUTs

4 states, 12 LUTs

async versus sync representations for sequential composition



8 states, 26 LUTs

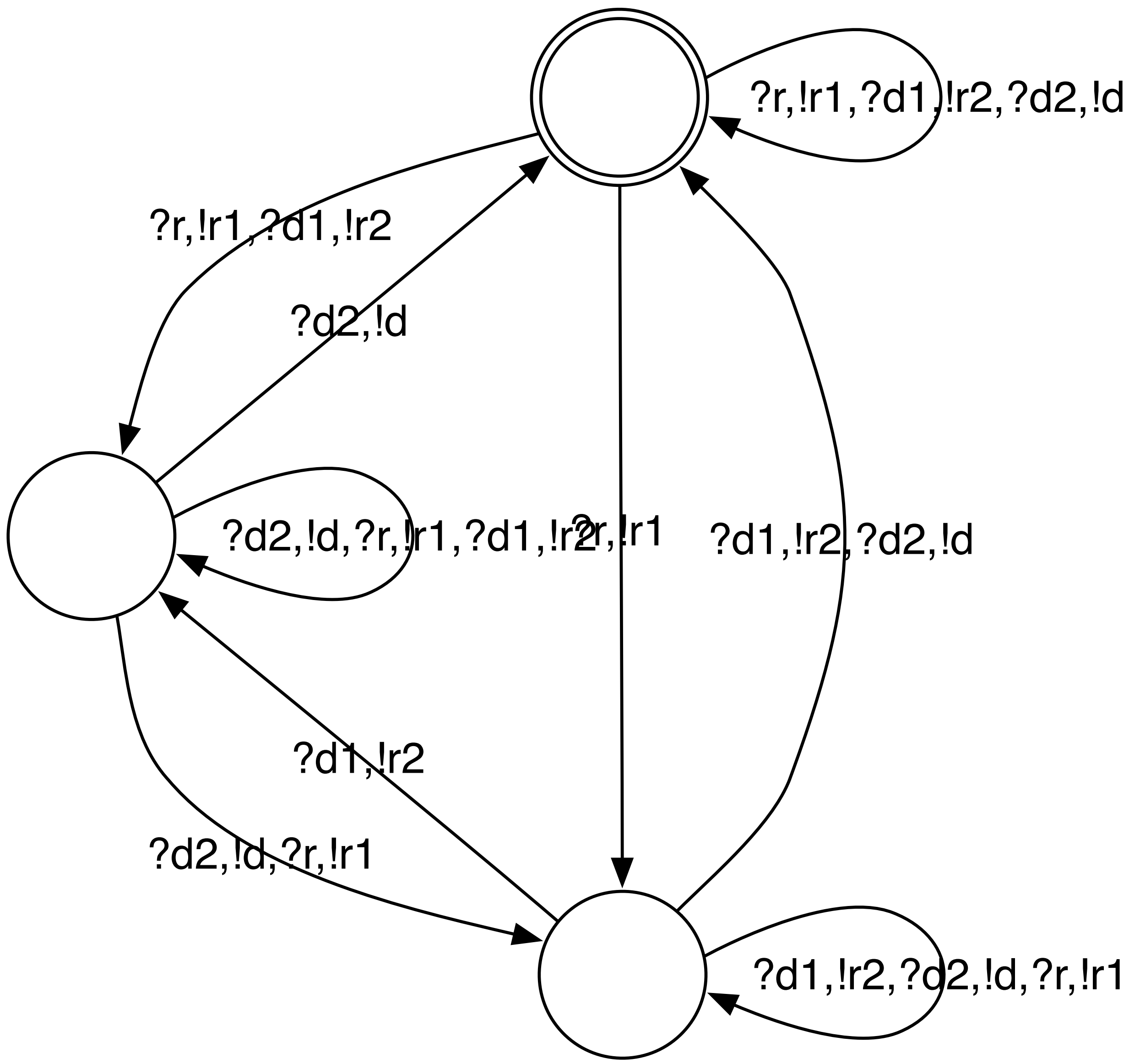


4 states, 12 LUTs

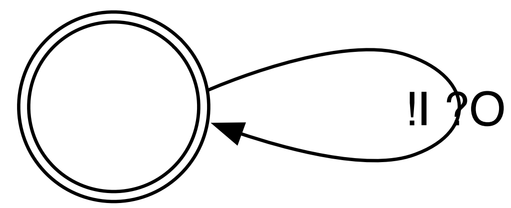
async versus sync representations for sequential composition

what is going on with sequential composition?

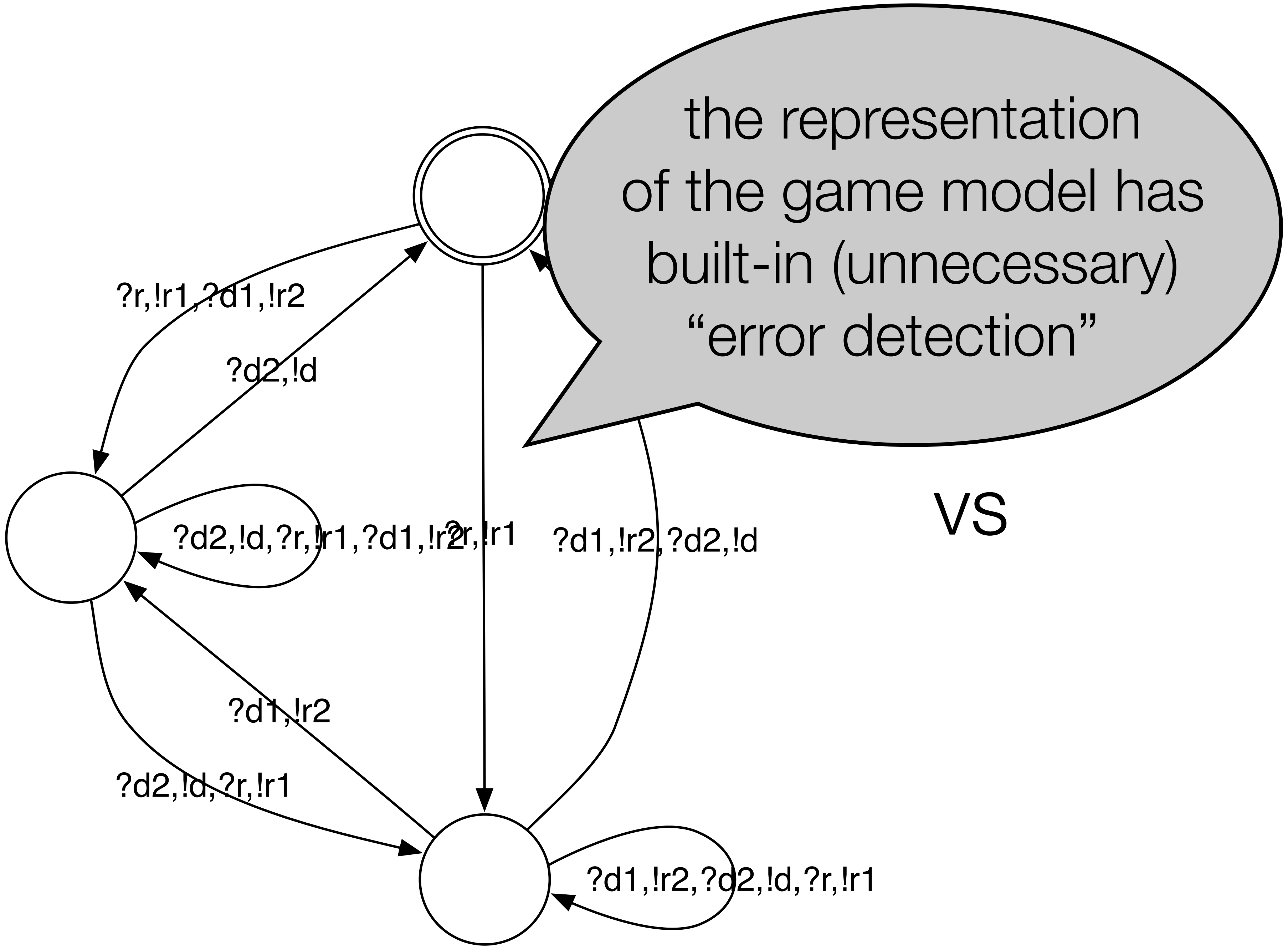
what is going on with sequential composition?



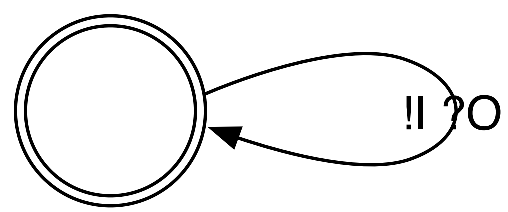
VS



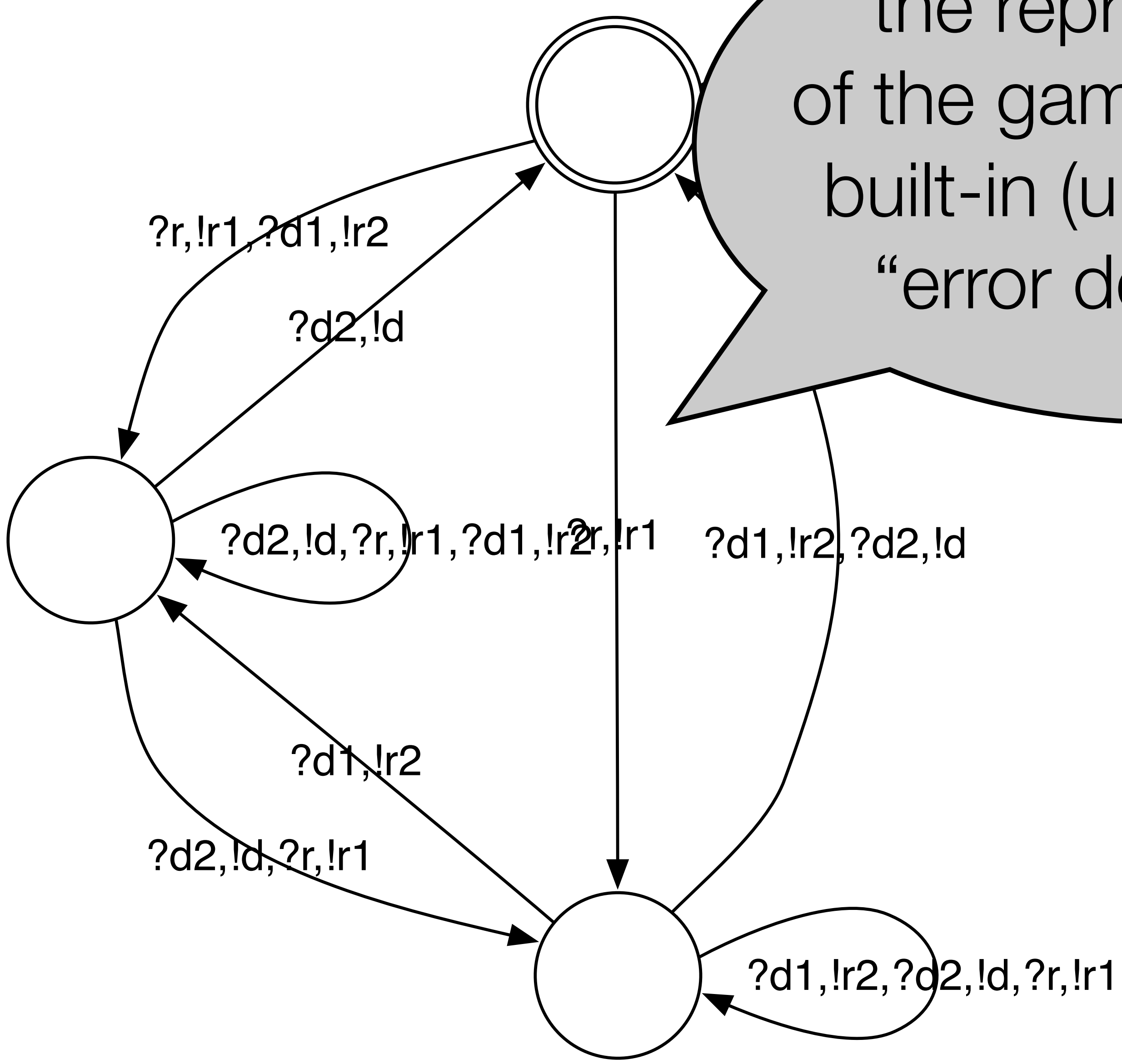
what is going on with sequential composition?



VS



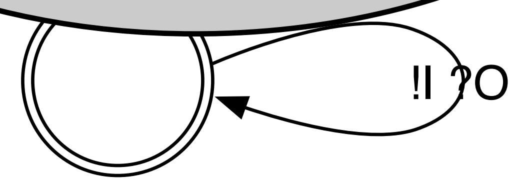
what is going on with sequential composition?



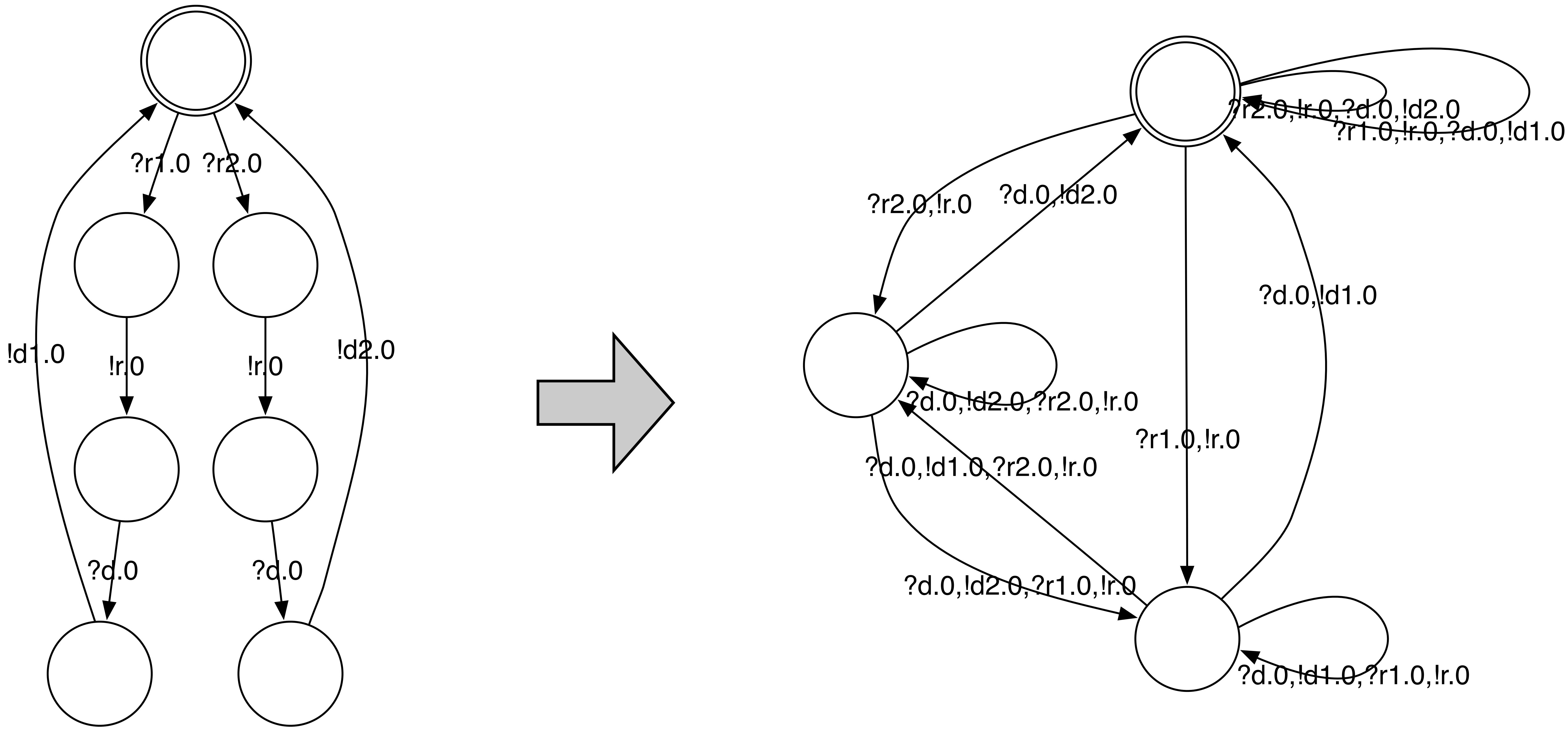
the representation of the game model built-in (unneces… "error detectio…

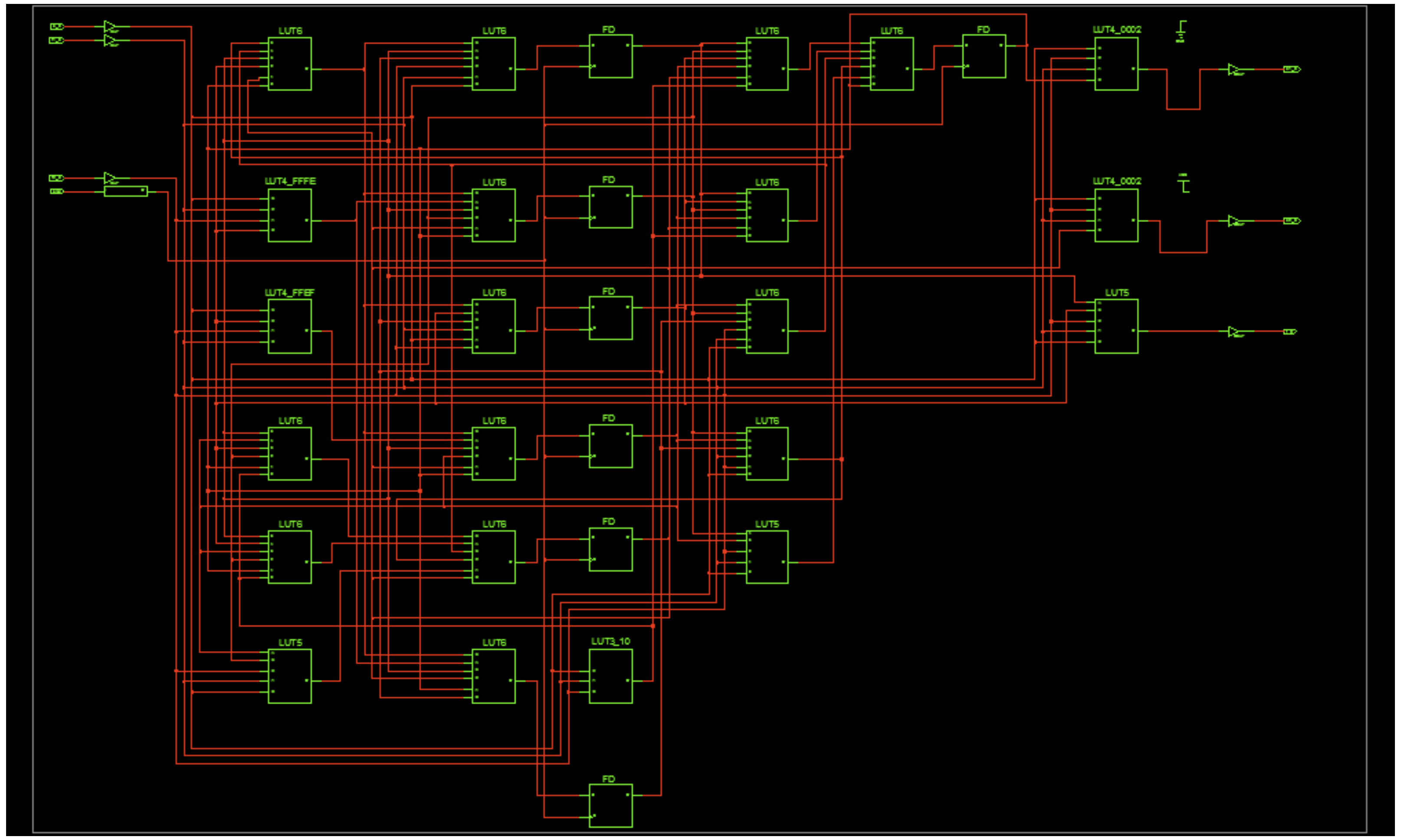
the behaviour of these two automata in legal environments is actually the same

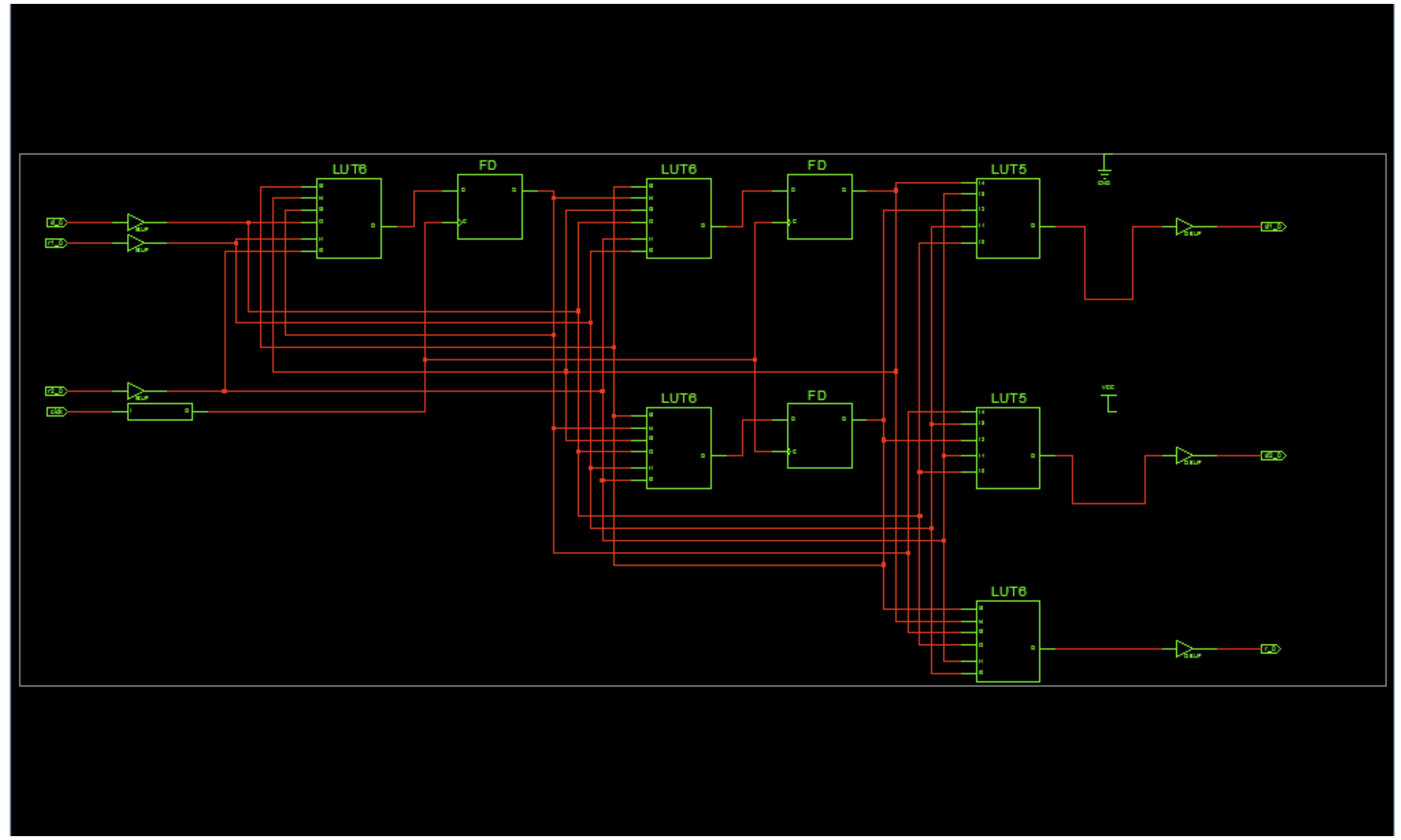
VS



a genuine application: diagonals (*bsci*)





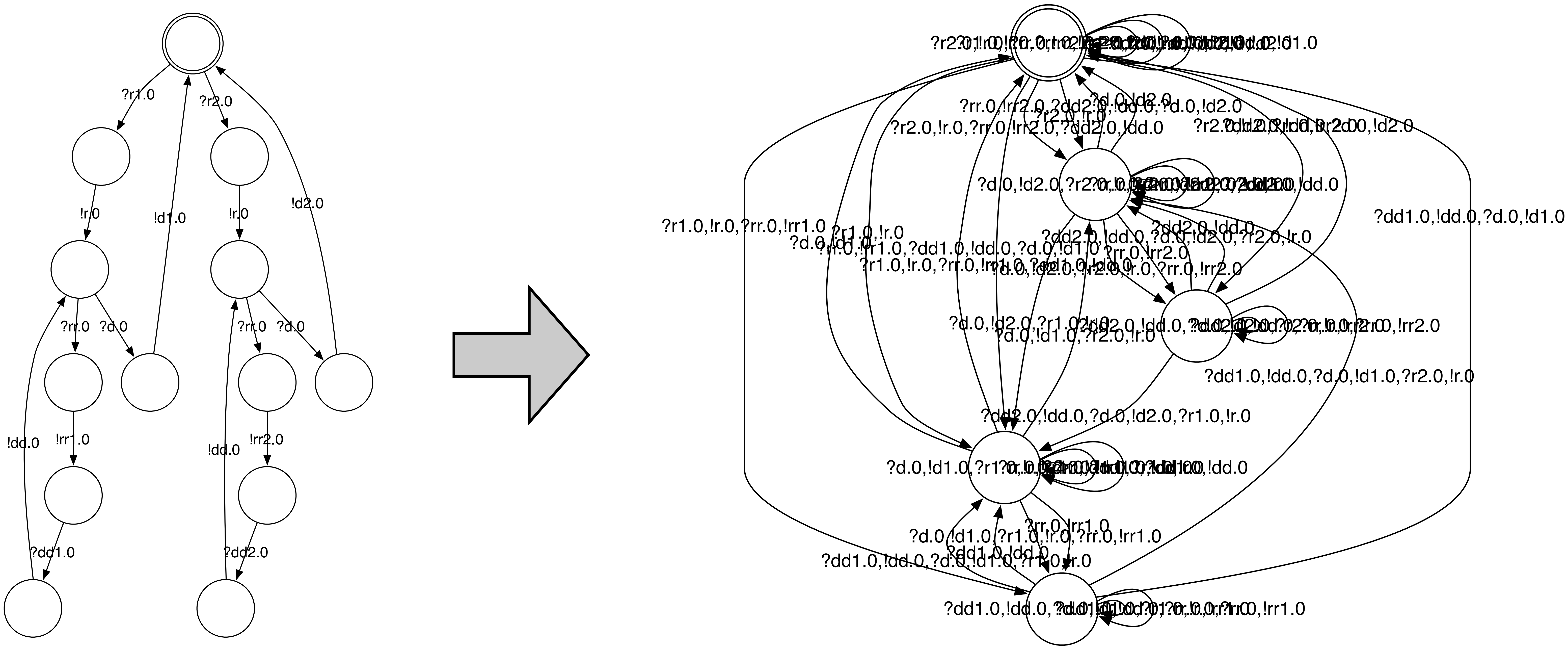


7 registers 22 LUTs

3 registers 6 LUTs

asynchronous vs. synchronous diagonals on *com*

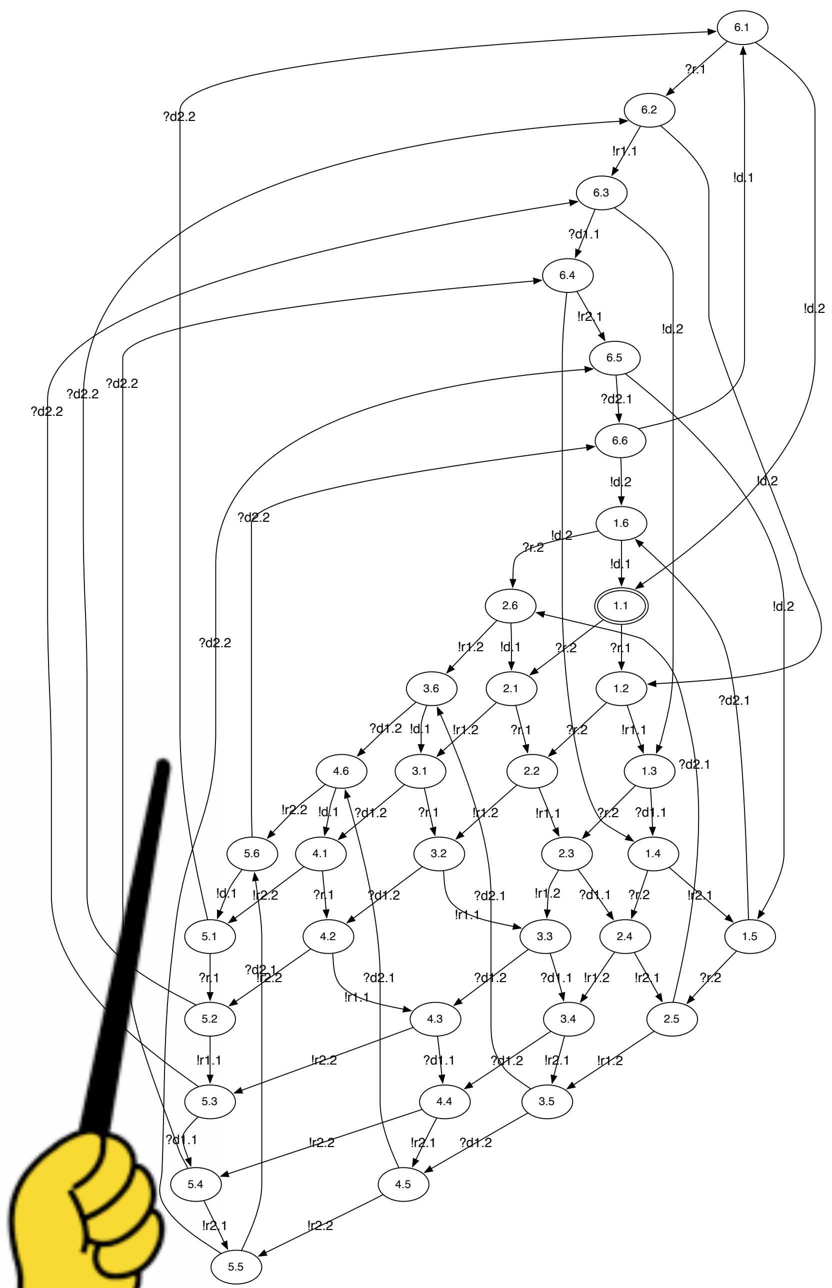
diagonal for `com` ⇒ `com` (*bsci*)



13 registers 94 LUTs

7 registers 77 LUTs

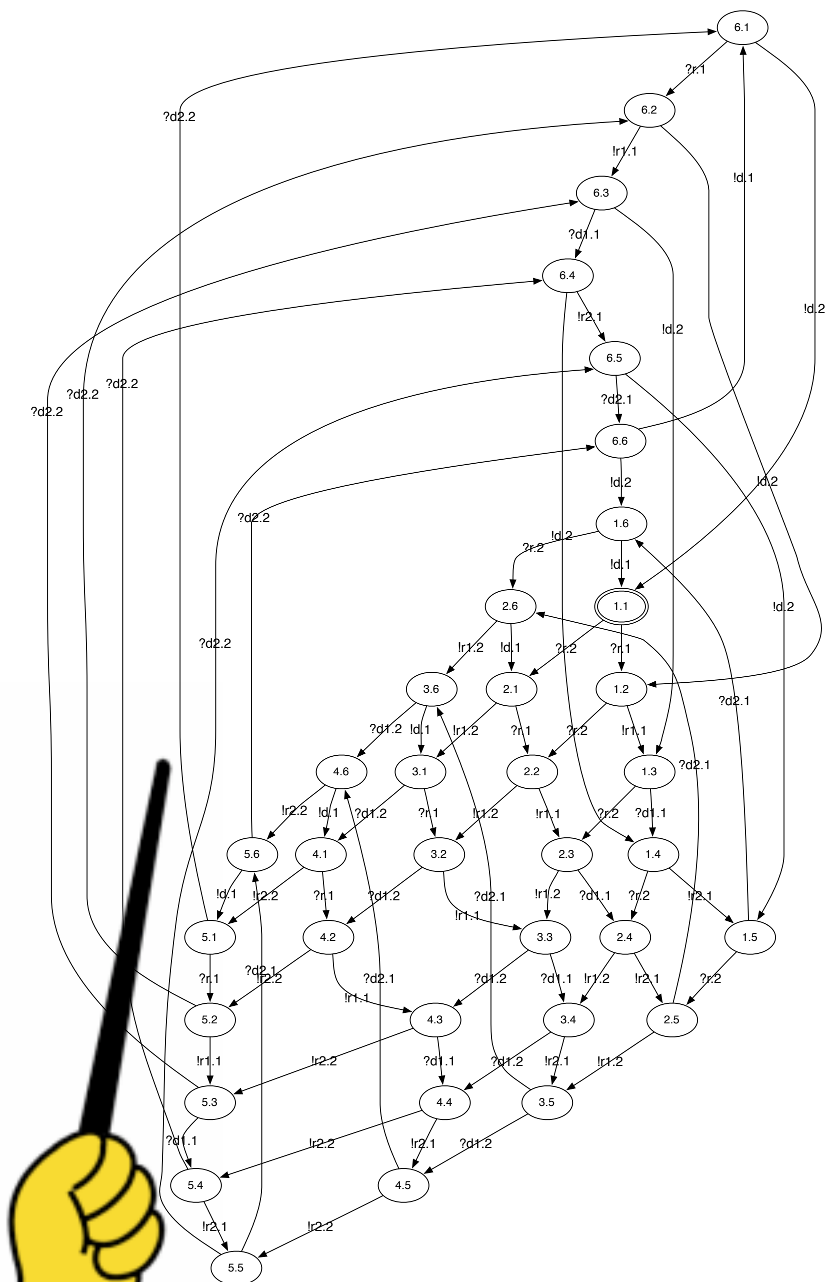
how about concurrent sharing? `seq ⊗ seq`

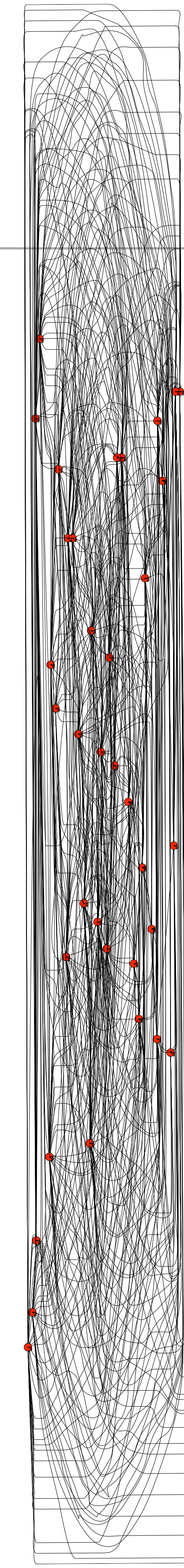


72 trans
36 states

how about concurrent sharing? `seq ⊗ seq`

72 trans
36 states





34 states
642 trans
(fails synthesis)

conclusion

conclusion

- it is nice to be able to reuse game models

conclusion

- it is nice to be able to reuse game models
    - but naive representation of game model is very inefficient

conclusion

- it is nice to be able to reuse game models
    - but naive representation of game model is very inefficient
- synchronous representation of (asynchronous) game model can be done via round abstraction

conclusion

- it is nice to be able to reuse game models
    - but naive representation of game model is very inefficient
- synchronous representation of (asynchronous) game model can be done via round abstraction
    - usually results in smaller circuits (always faster)

conclusion

- it is nice to be able to reuse game models
    - but naive representation of game model is very inefficient
- synchronous representation of (asynchronous) game model can be done via round abstraction
    - usually results in smaller circuits (always faster)
    - still room for optimisation (eliminate “error detection”)

conclusion

- it is nice to be able to reuse game models
    - but naive representation of game model is very inefficient
- synchronous representation of (asynchronous) game model can be done via round abstraction
    - usually results in smaller circuits (always faster)
    - still room for optimisation (eliminate “error detection”)
- r.a. can be applied to any game model, not just to game models of constants

conclusion

- it is nice to be able to reuse game models
    - but naive representation of game model is very inefficient
- synchronous representation of (asynchronous) game model can be done via round abstraction
    - usually results in smaller circuits (always faster)
    - still room for optimisation (eliminate “error detection”)
- r.a. can be applied to any game model, not just to game models of constants
    - peep-hole optimisation for games

conclusion

- it is nice to be able to reuse game models
    - but naive representation of game model is very inefficient
- synchronous representation of (asynchronous) game model can be done via round abstraction
    - usually results in smaller circuits (always faster)
    - still room for optimisation (eliminate “error detection”)
- r.a. can be applied to any game model, not just to game models of constants
    - peep-hole optimisation for games
- concurrent sharing not feasible