EPSRC Engineering and Physical Sciences Research Council

Newcastle University

Generalised Asynchronous Arbiter

Stanislavs Golubcovs, **Andrey Mokhov**, Alex Bystrov, Danil Sokolov, Alex Yakovlev

27 June 2019, Aachen

EPSRC

Engineering and Physical Sciences Research Council

Newcastle University

A tour to Arbiterland!

Generalised Asynchronous Arbiter

Stanislavs Golubcovs, **Andrey Mokhov**, Alex Bystrov, Danil Sokolov, Alex Yakovlev

27 June 2019, Aachen

Why a new arbiter?

Why arbiters?

Arbiters orchestrate access to shared resources:

- Memory, where multiple processors meet
- Road intersections, where multiple cars meet
- Ice-cream shops, where multiple overheated people of Aachen meet

Why arbiters?

Arbiters orchestrate access to shared resources:

- Memory, where multiple processors meet
- Road intersections, where multiple cars meet
- Ice-cream shops, where multiple overheated people of Aachen meet

Good properties that we want from arbiters:

- Low latency: "I need an ice-cream **as soon as possible**, please!"

Why arbiters?

Arbiters orchestrate access to shared resources:

- Memory, where multiple processors meet
- Road intersections, where multiple cars meet
- Ice-cream shops, where multiple overheated people of Aachen meet

Good properties that we want from arbiters:

- Low latency: “I need an ice-cream **as soon as possible**, please!”
- Deadlock freedom: “What do you mean you **run out** of ice-cream?!”

Why arbiters?

Arbiters orchestrate access to shared resources:

- Memory, where multiple processors meet
- Road intersections, where multiple cars meet
- Ice-cream shops, where multiple overheated people of Aachen meet

Good properties that we want from arbiters:

- Low latency: “I need an ice-cream **as soon as possible**, please!”
- Deadlock freedom: “What do you mean you **run out** of ice-cream?!”
- Fairness: “Hey, that’s **my** ice-cream!”

Why arbiters?

Arbiters orchestrate access to shared resources:

- Memory, where multiple processors meet
- Road intersections, where multiple cars meet
- Ice-cream shops, where multiple overheated people of Aachen meet

Good properties that we want from arbiters:

- Low latency: “I need an ice-cream **as soon as possible**, please!”
- Deadlock freedom: “What do you mean you **run out** of ice-cream?!”
- Fairness: “Hey, that’s **my** ice-cream!”
- Constraints: “Can I have **one pistachio and one non-vanilla** scoop?”

Why a new arbiter?

Arbiters orchestrate access to shared resources:

- Memory, where multiple processors meet
- Road intersections, where multiple cars meet
- Ice-cream shops, where multiple overheated people of Aachen meet

Good properties that we want from arbiters:

- Low latency: **asynchronous request capture, event-driven**
- Deadlock freedom: **formally verified using Petri nets**
- Fairness:
- Constraints:

} **generalised decision making by combinational logic**

What's the main challenge?

What is the main challenge?

What we want

What is the main challenge?

What we have

What we want

What is the main challenge?

Design, synthesis

What we have

What we want

What is the main challenge?

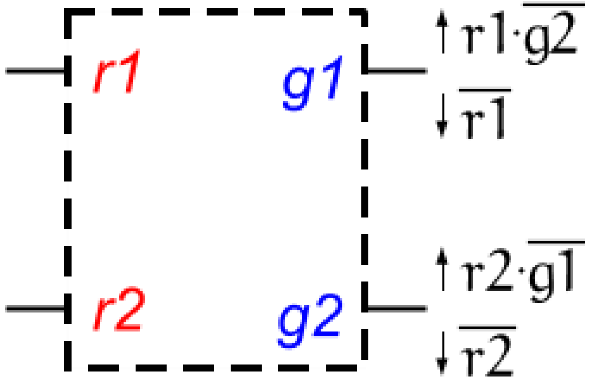
Design, ~~synthesis~~

What we have

What we want

What is the main challenge?

Mutex gate



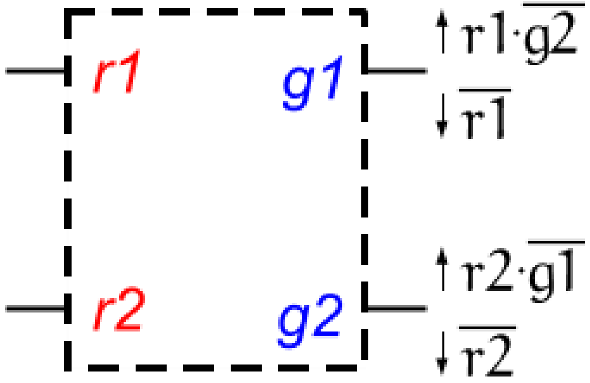
What we have

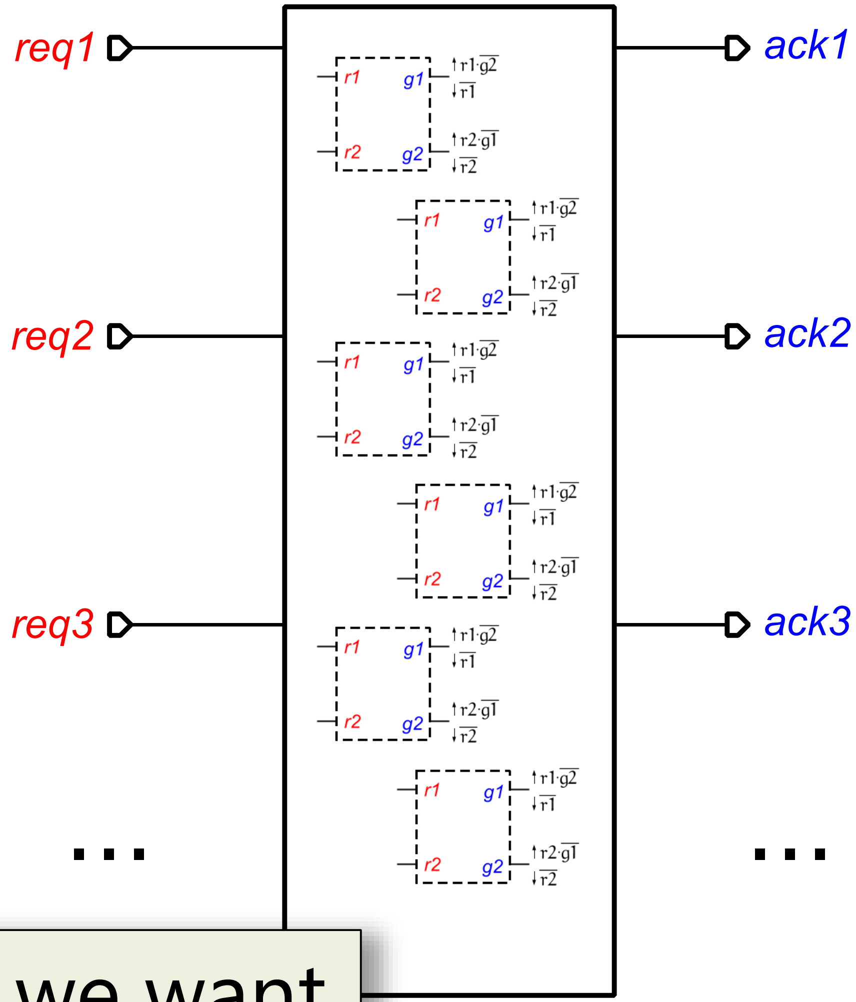
What we want

What is the main challenge?

Complex arbiter

Mutex gate

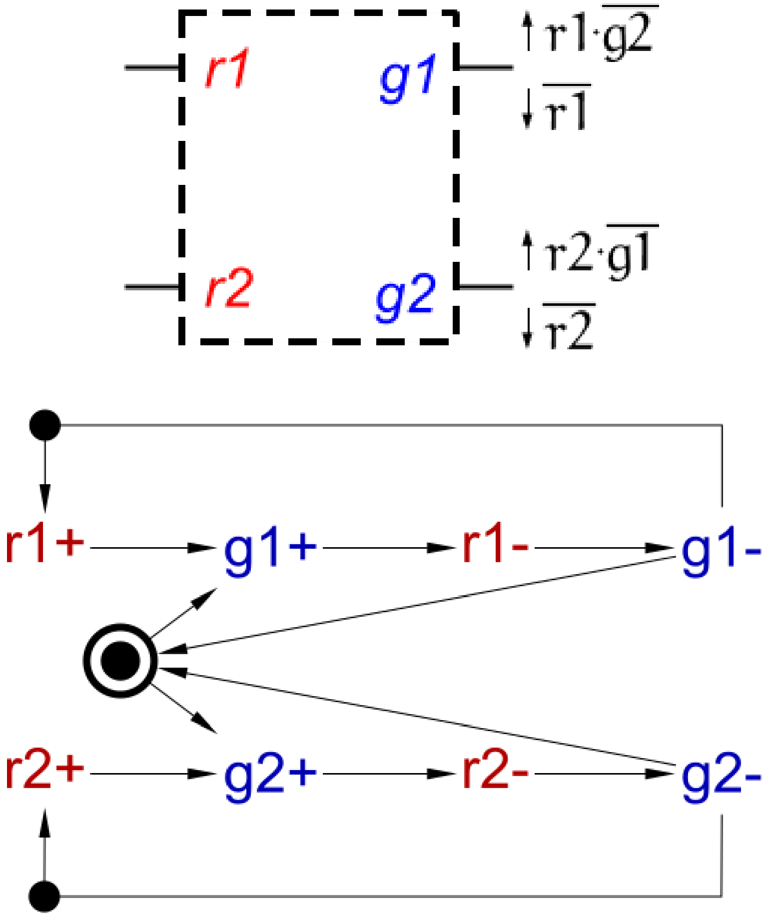




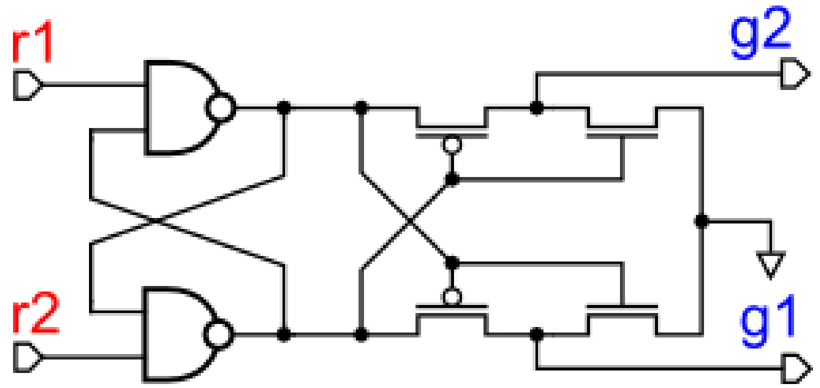
What we have

What we want

Mutual exclusion (mutex) element



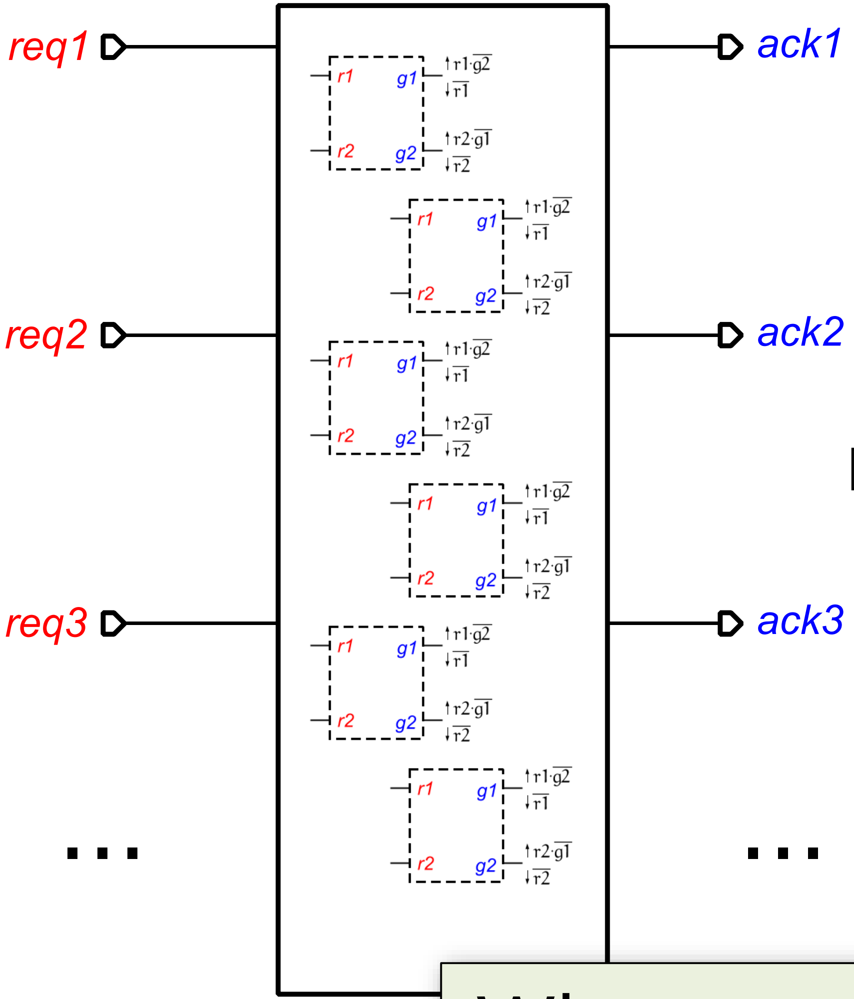
Circuit and PN specification



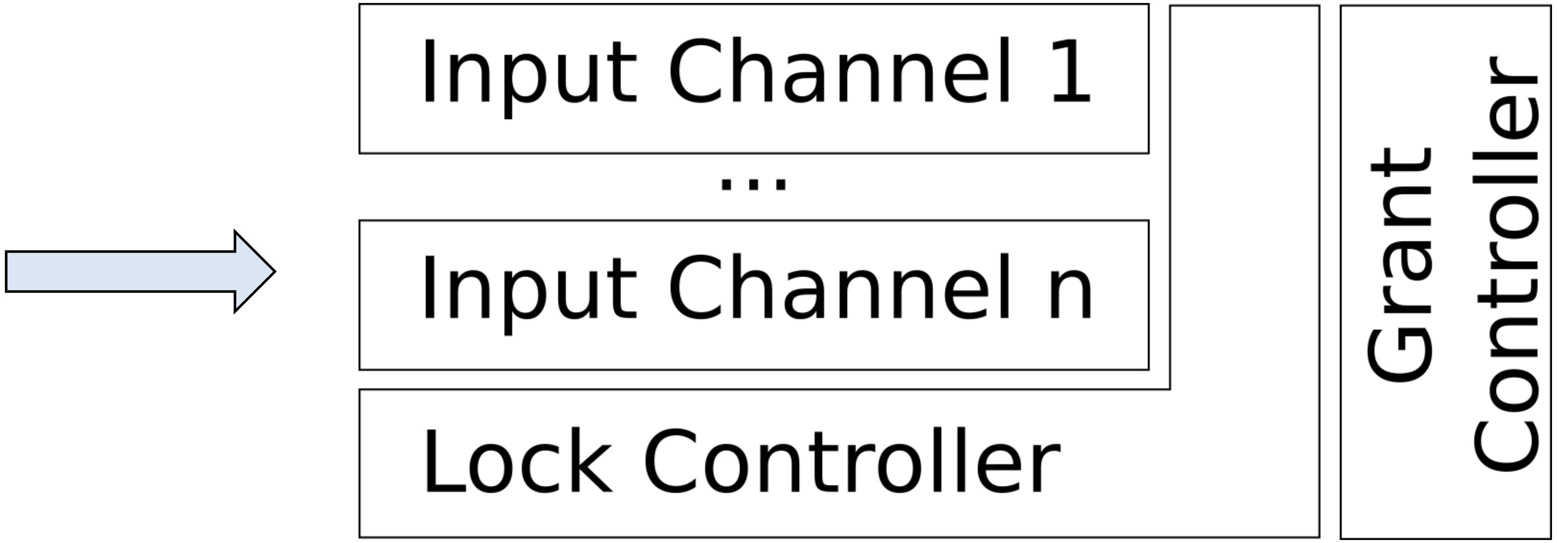
Standard implementation

The main idea

The main idea

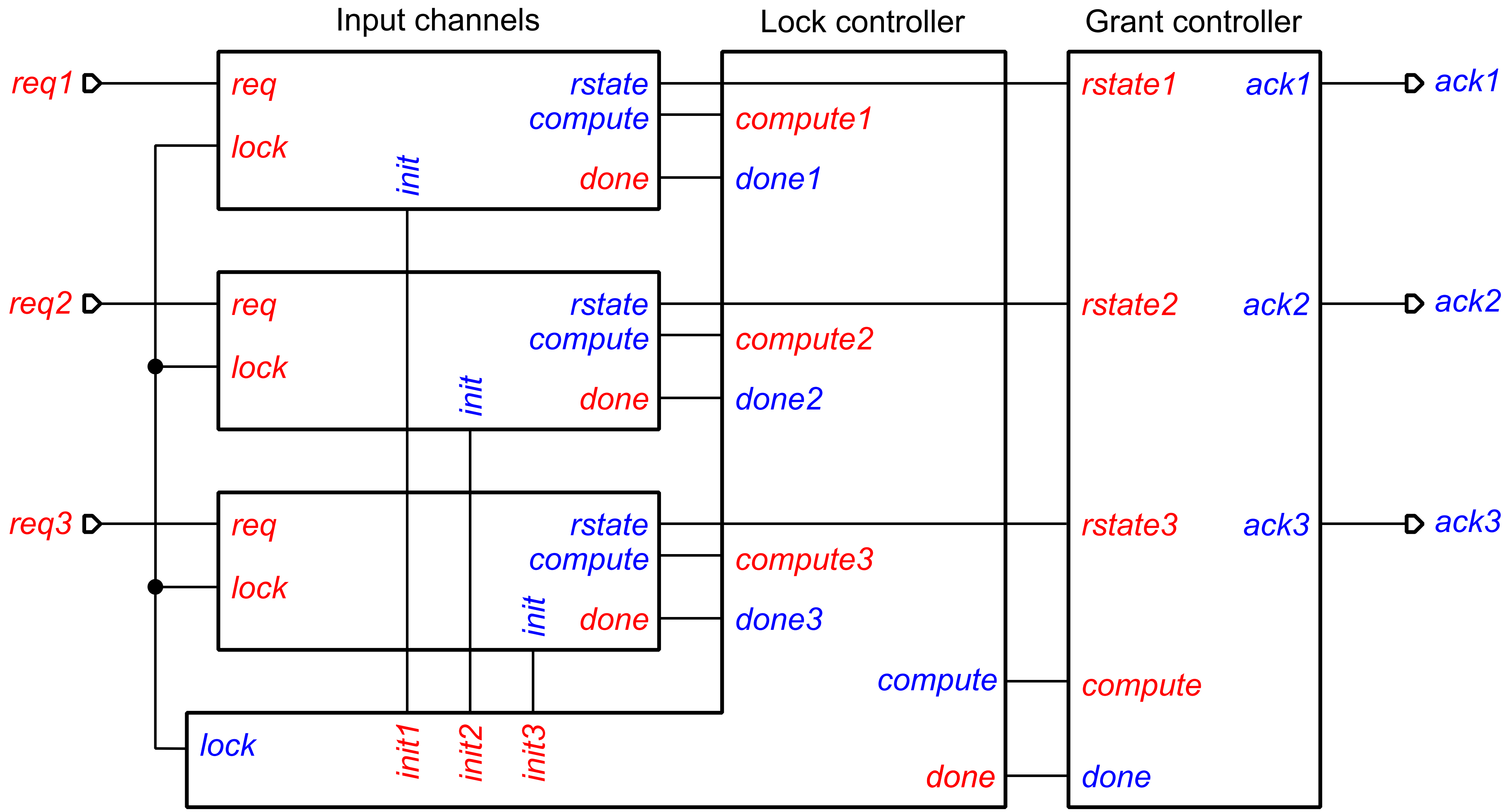


What we want

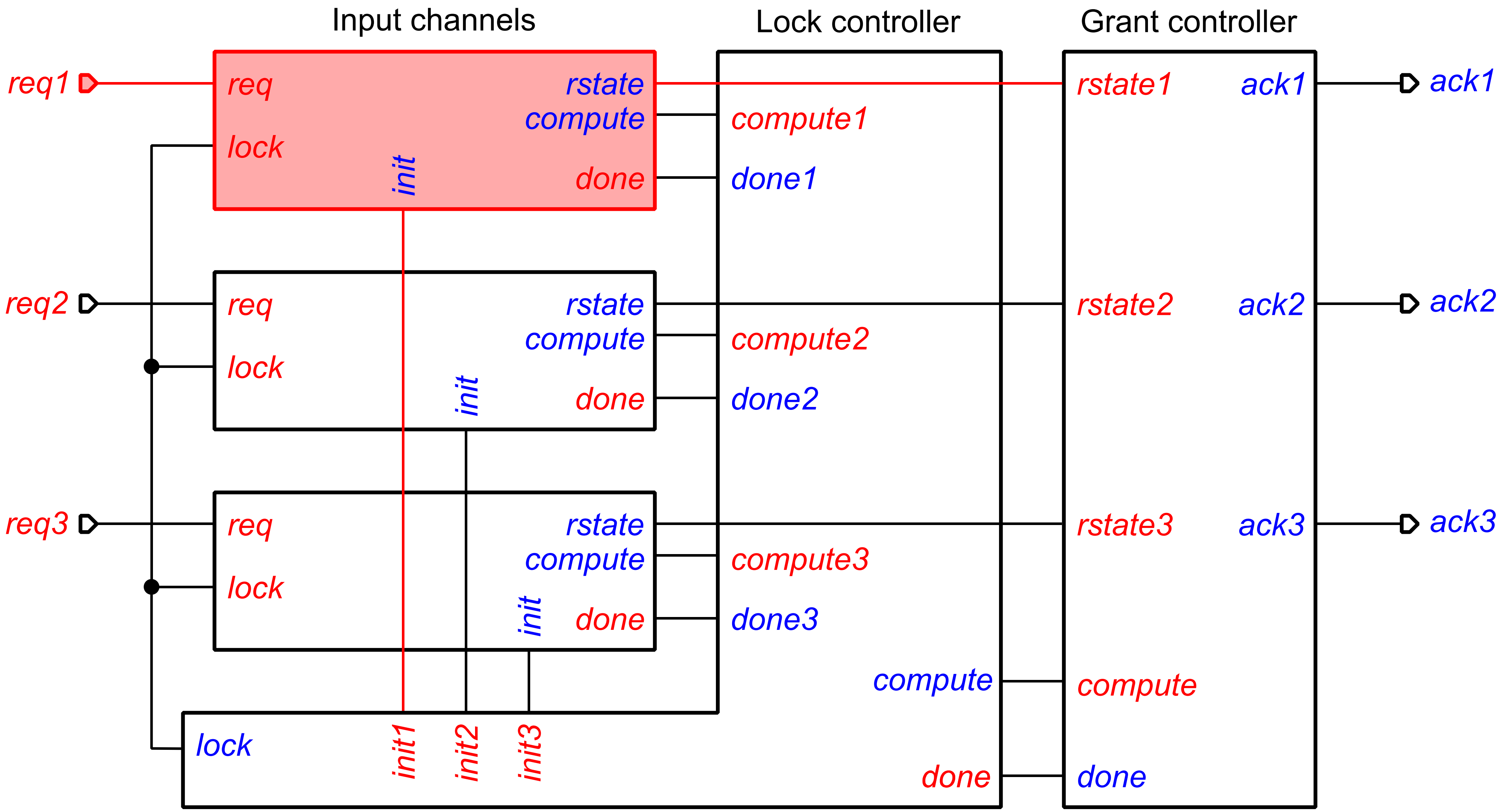


Top-level structure

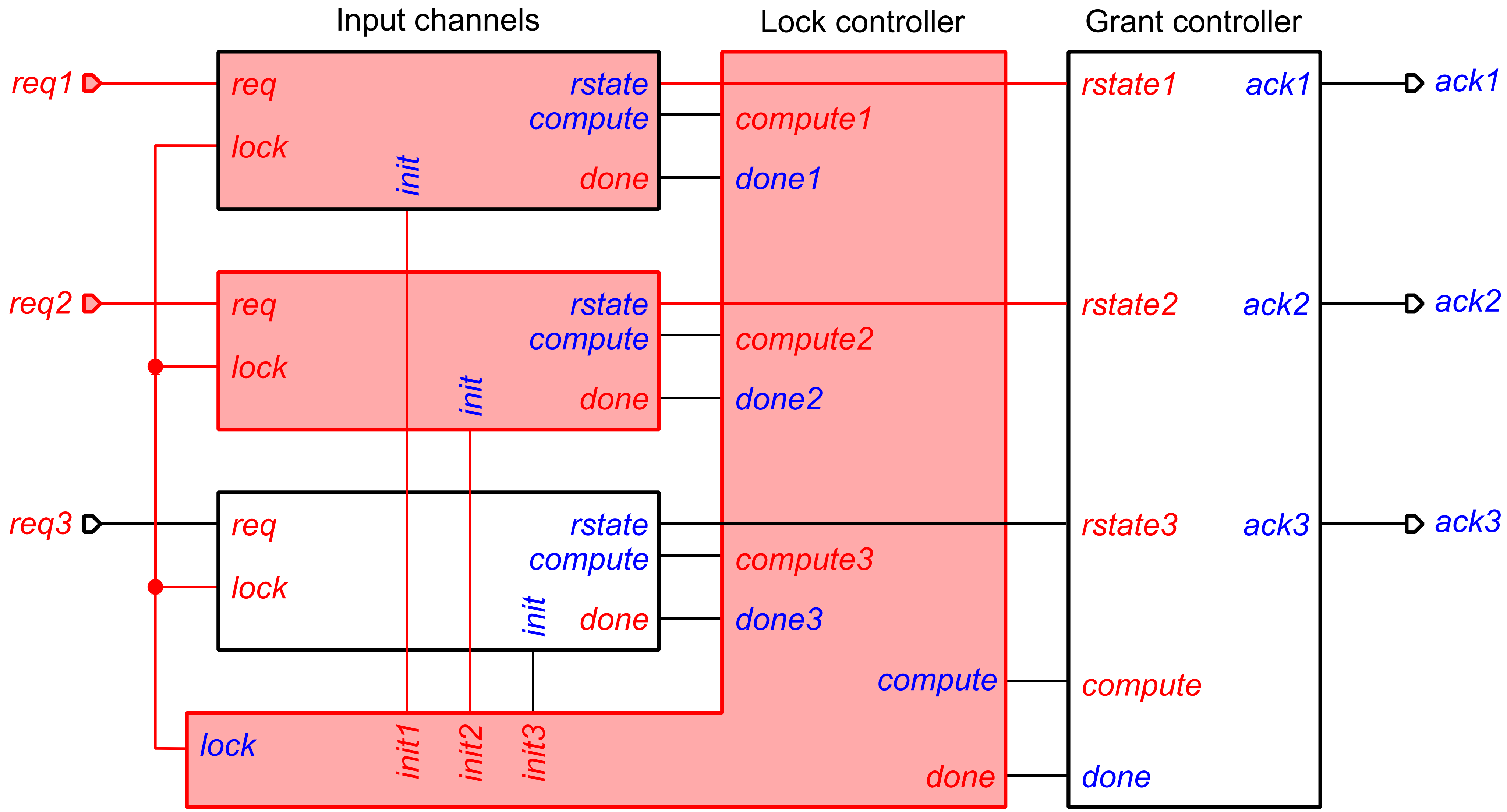
Initial state



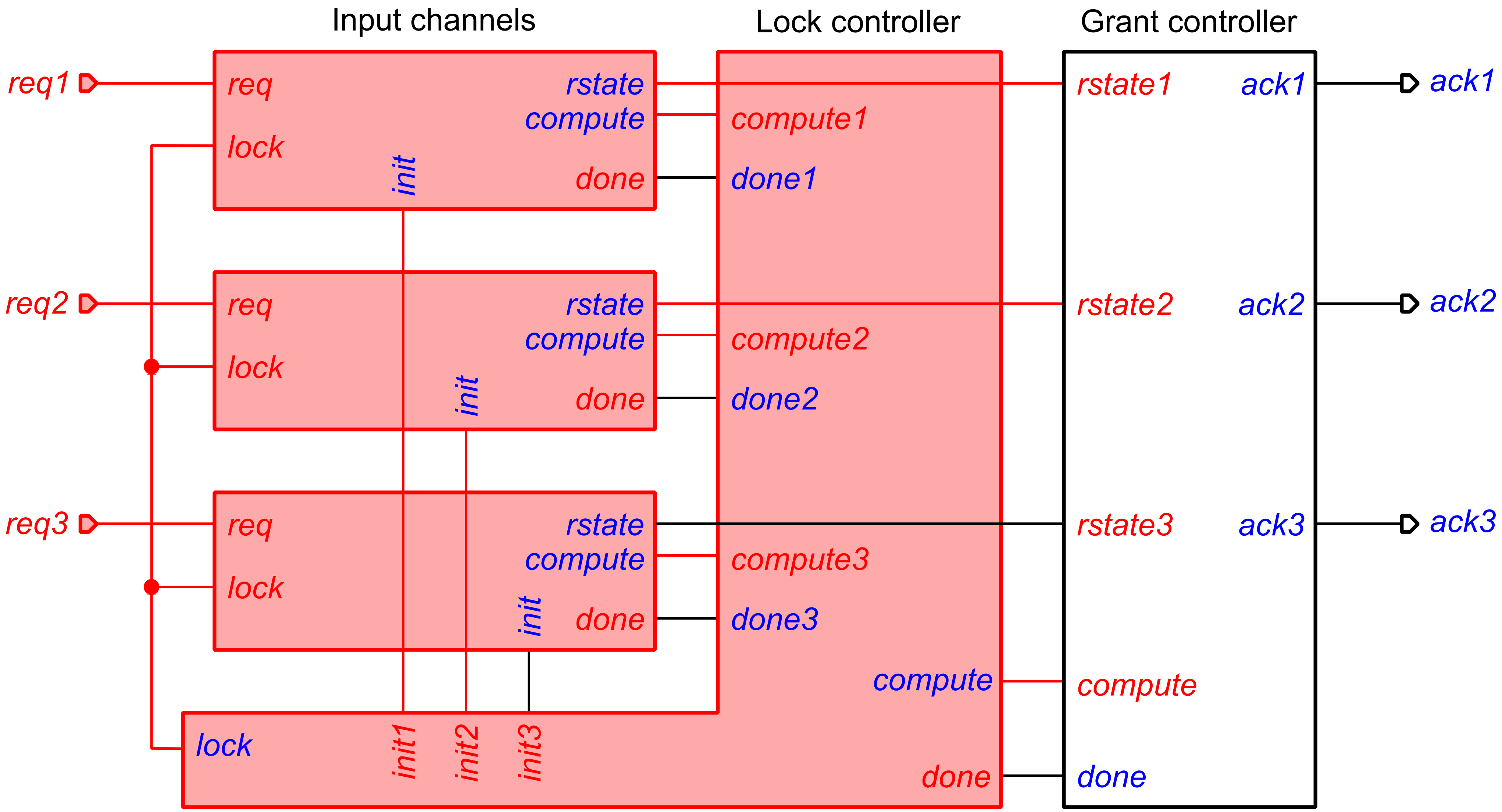
Request 1 arrives, activates the Lock



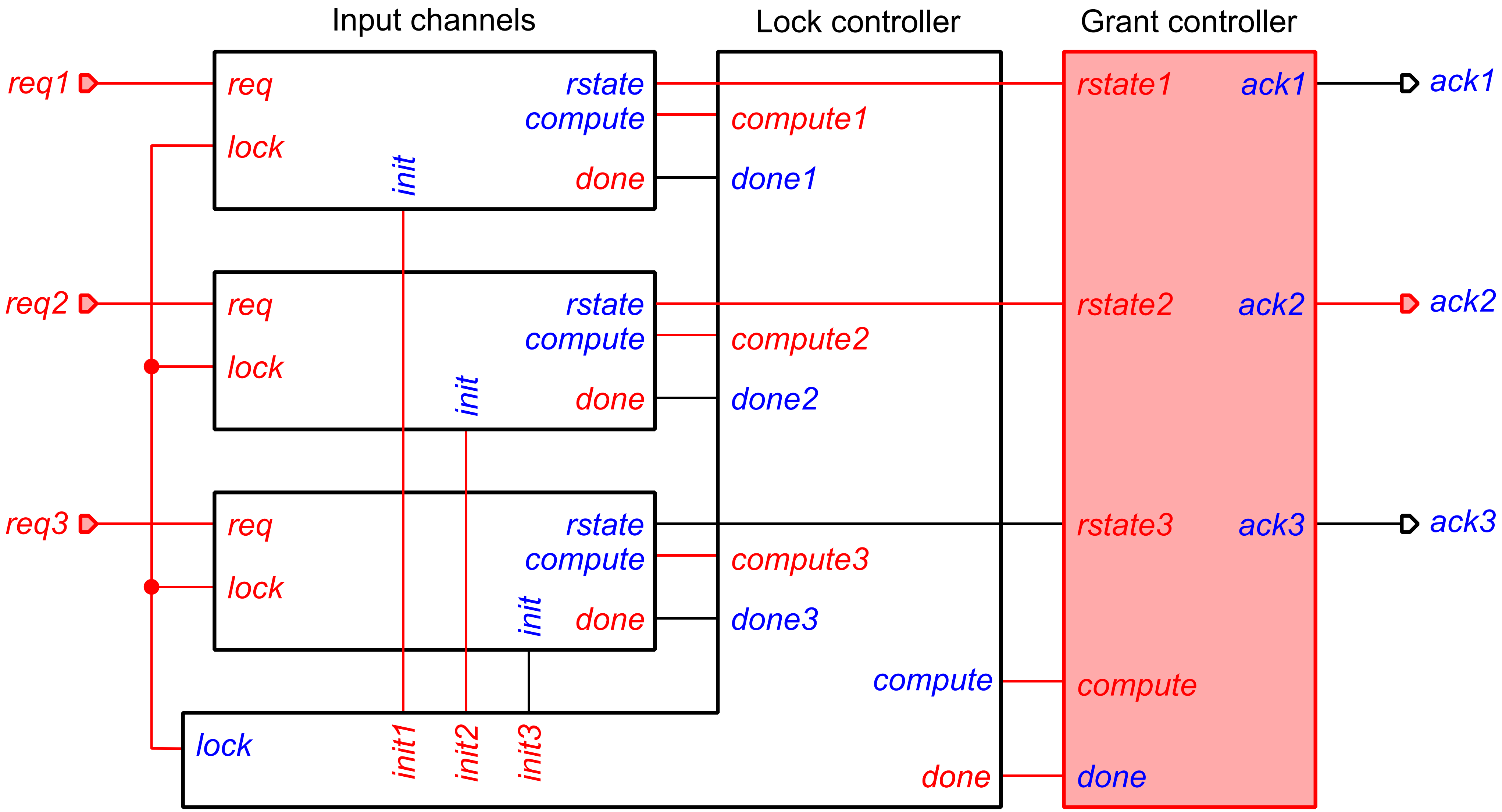
Request 2 arrives, just in time to go through



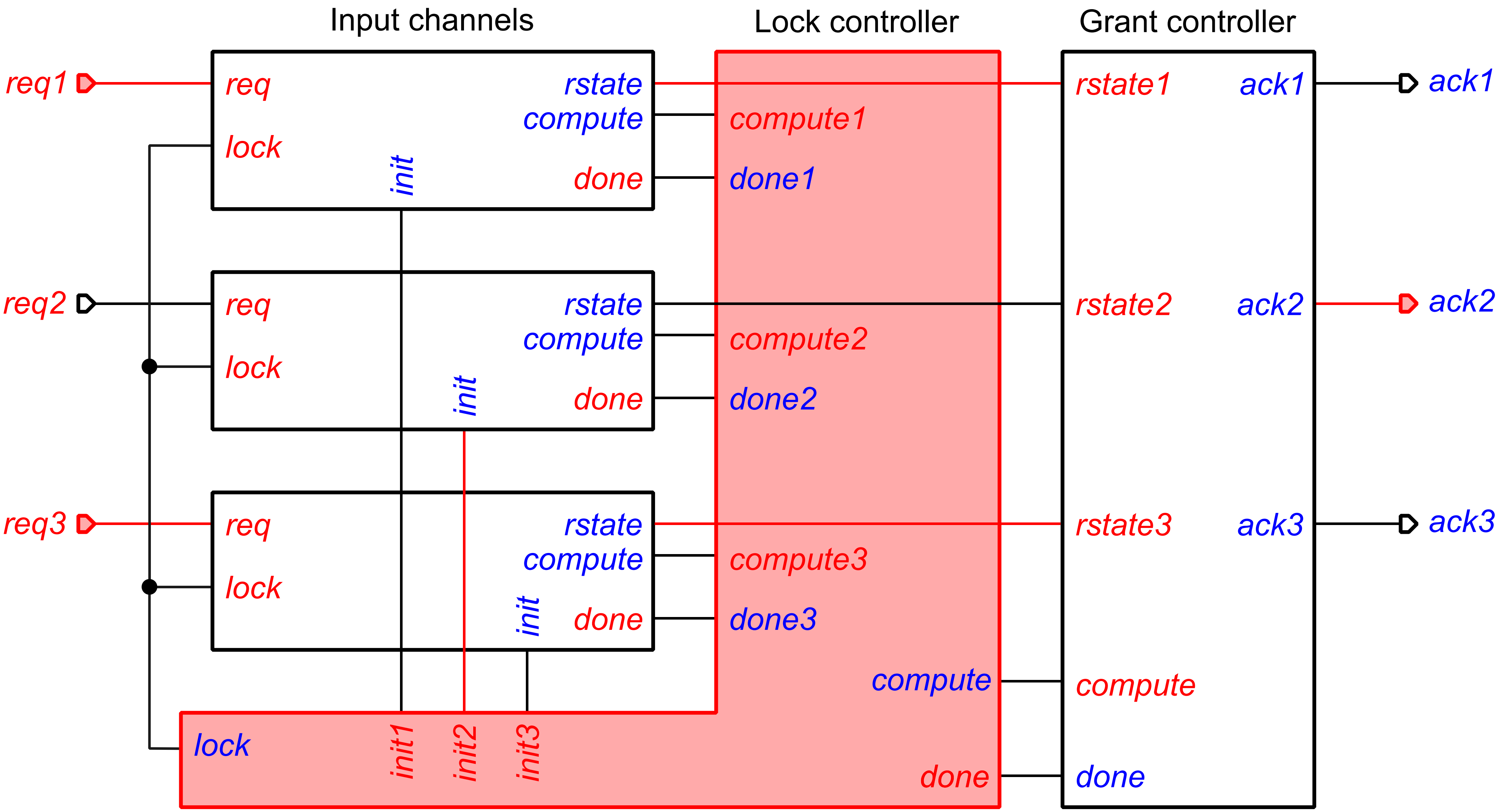
Request 3 arrives too late



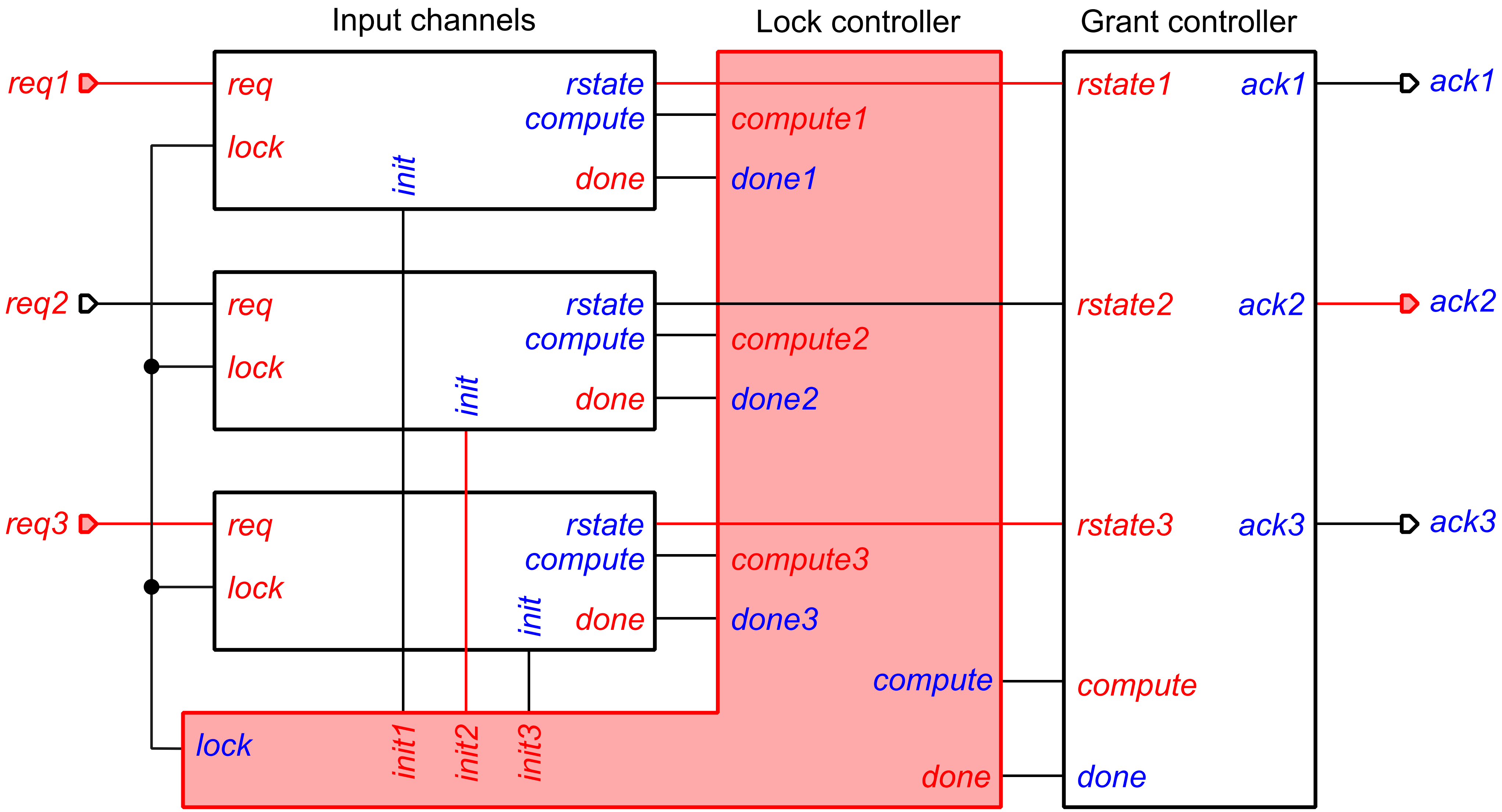
Make decision, grant request 2



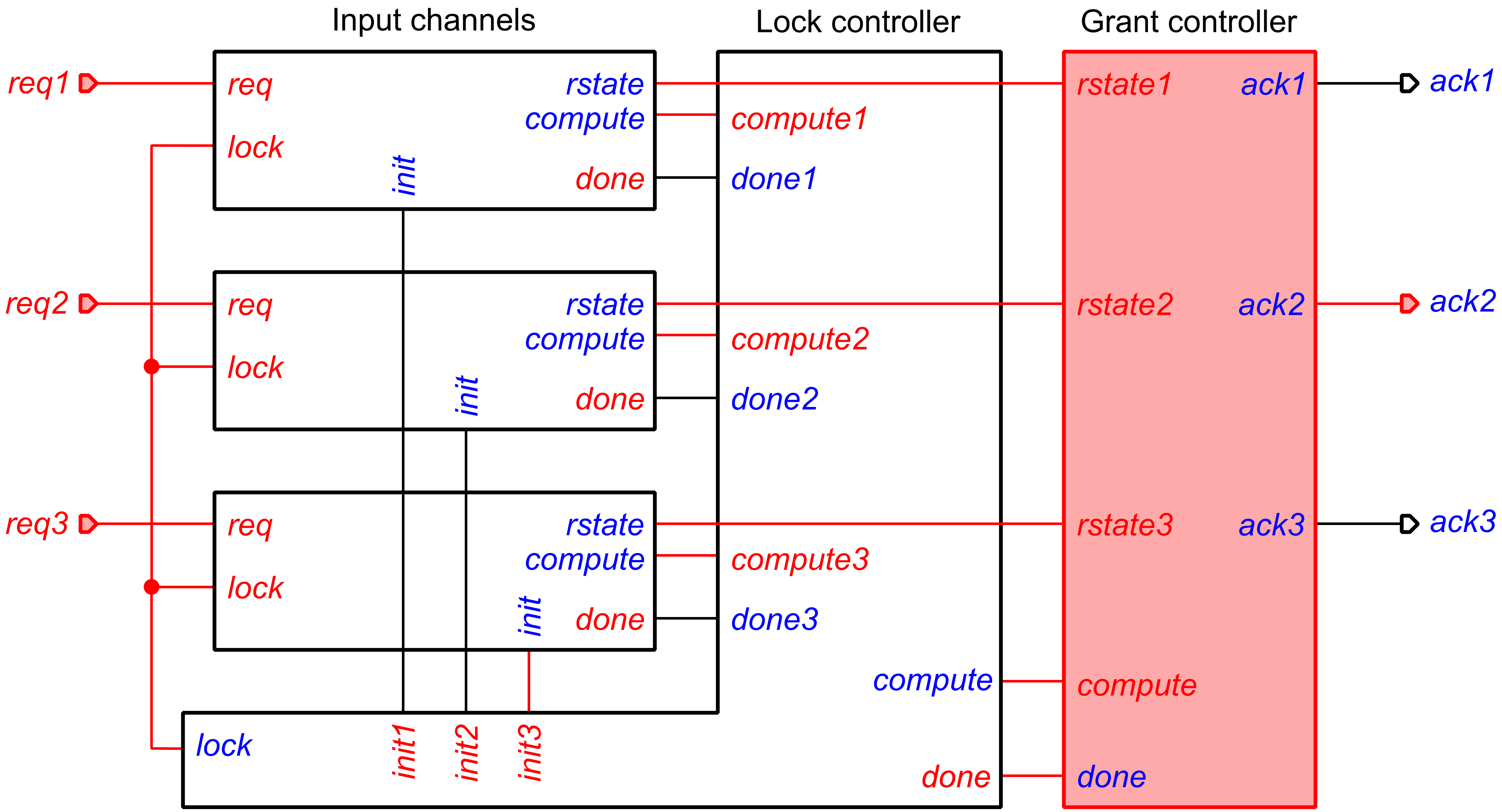
Release the Lock, allowing Request 3 in



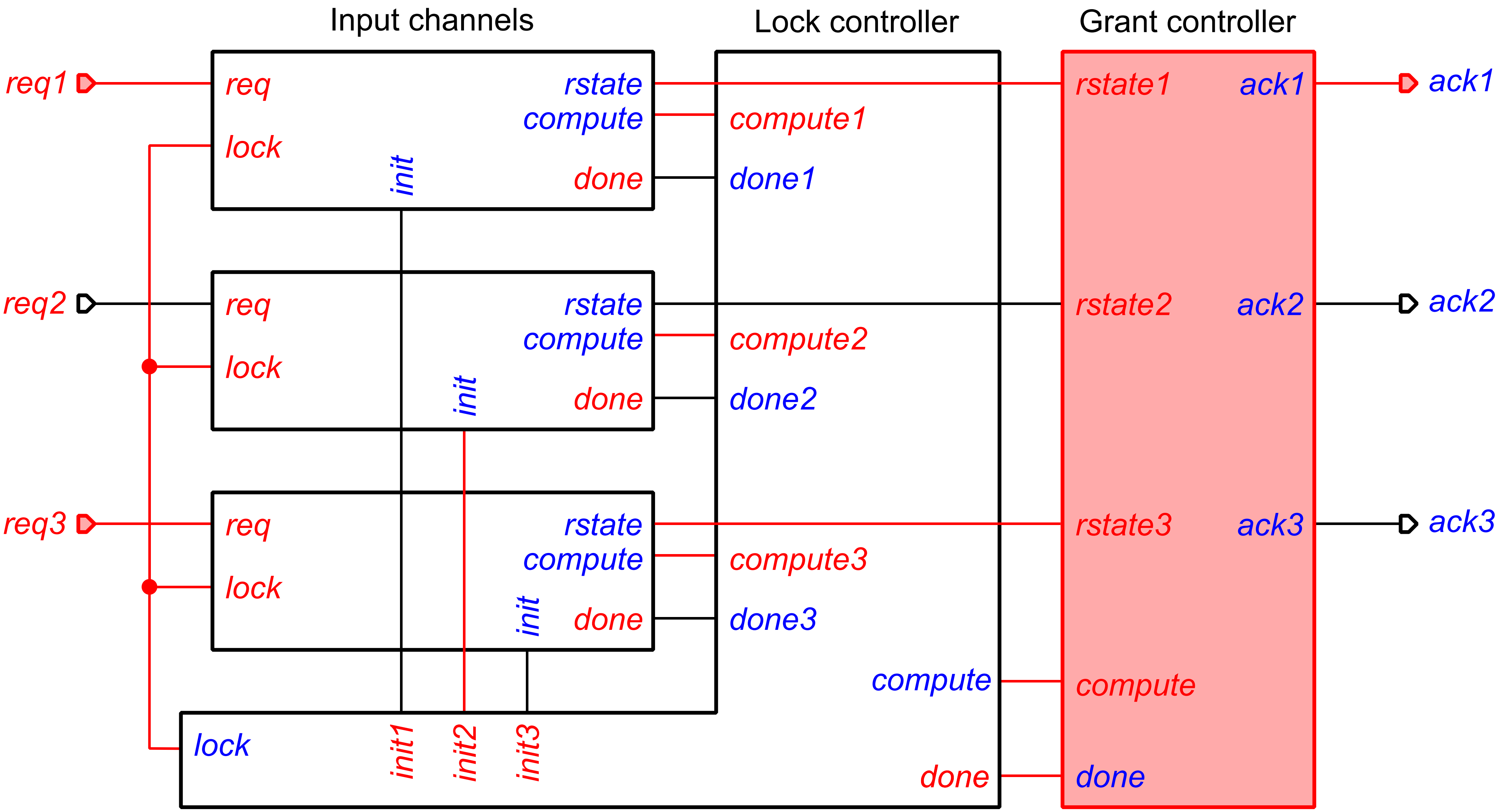
Request 2 is released, but too late



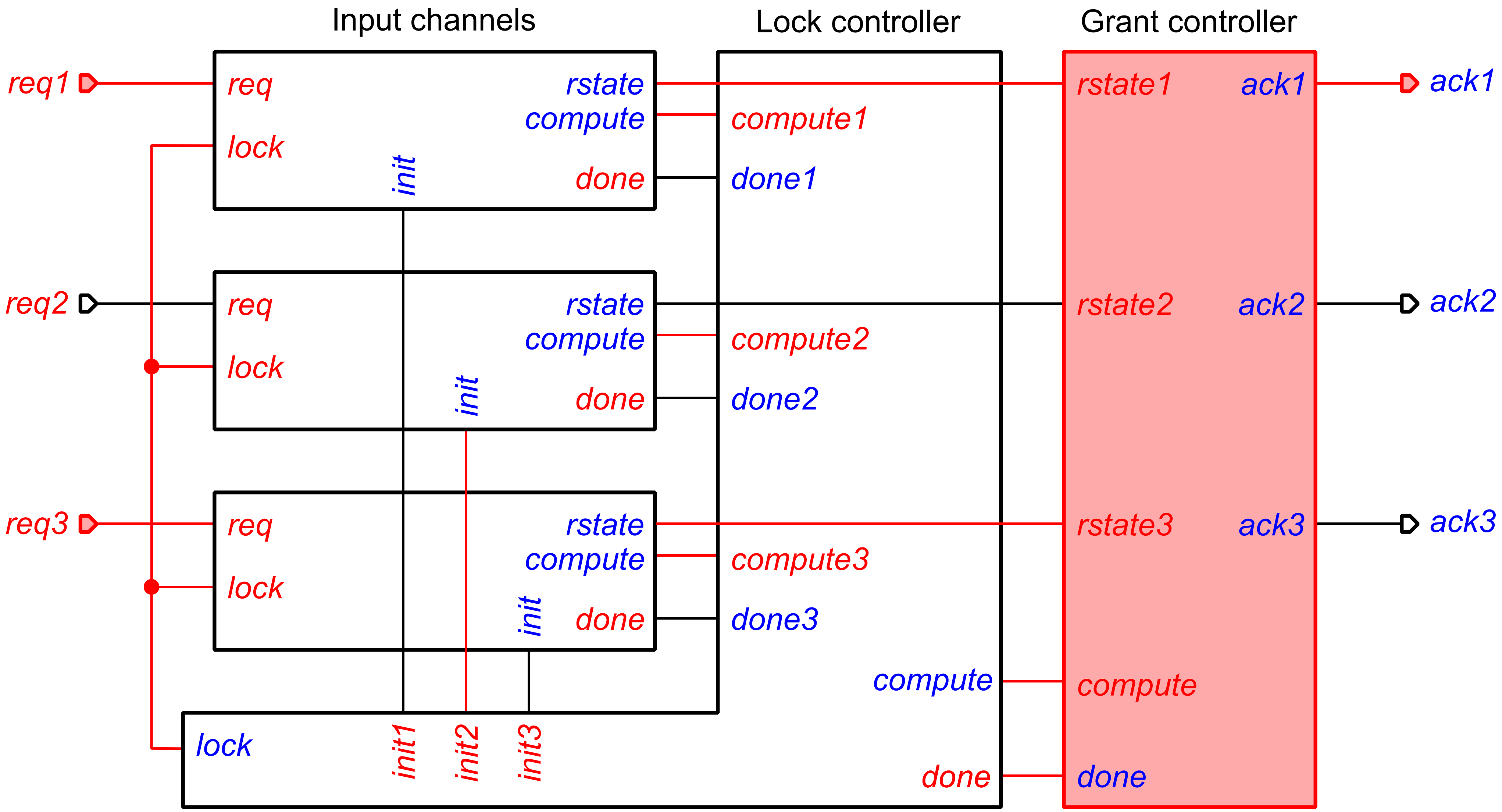
Grant controller decides there is nothing to do



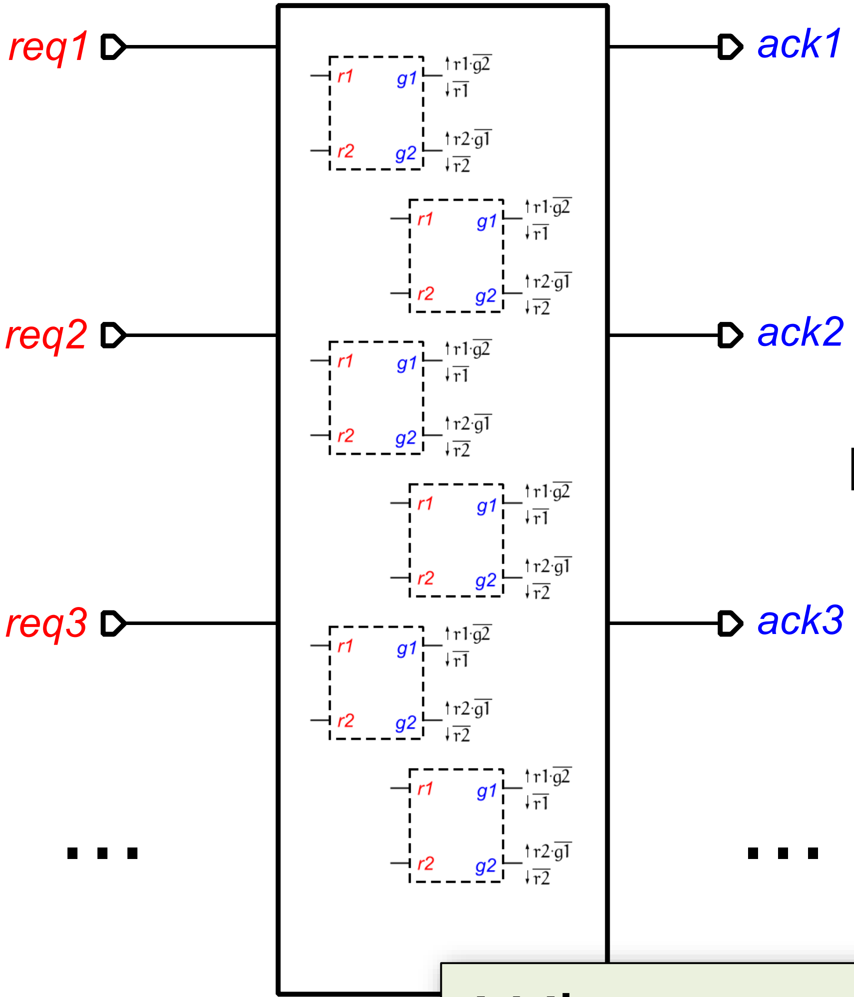
Lock is released; falling Request 2 goes through



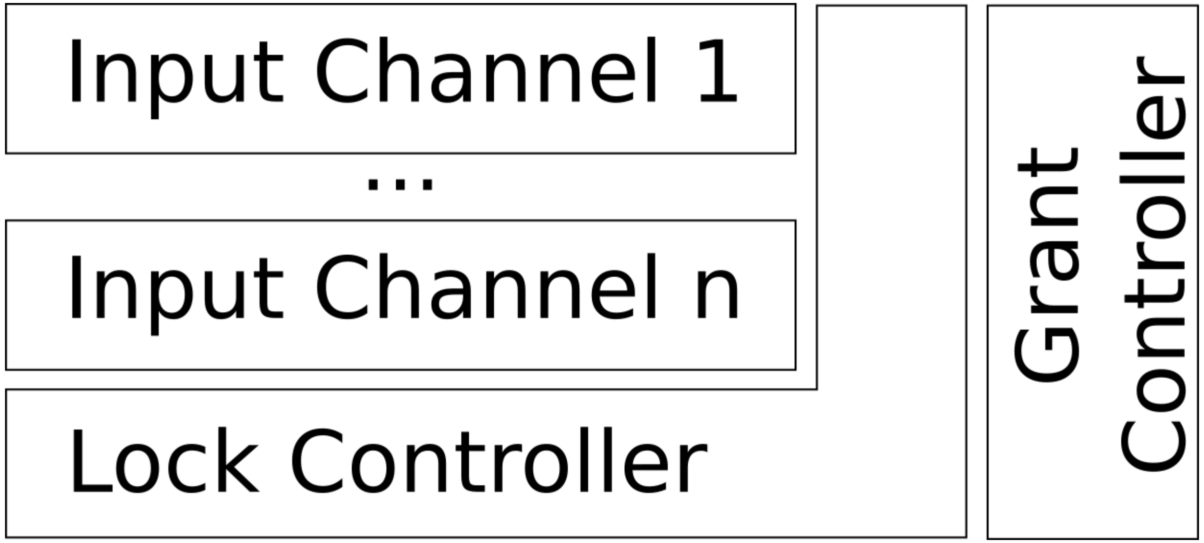
Request 1 is (finally) granted



The main idea



What we want

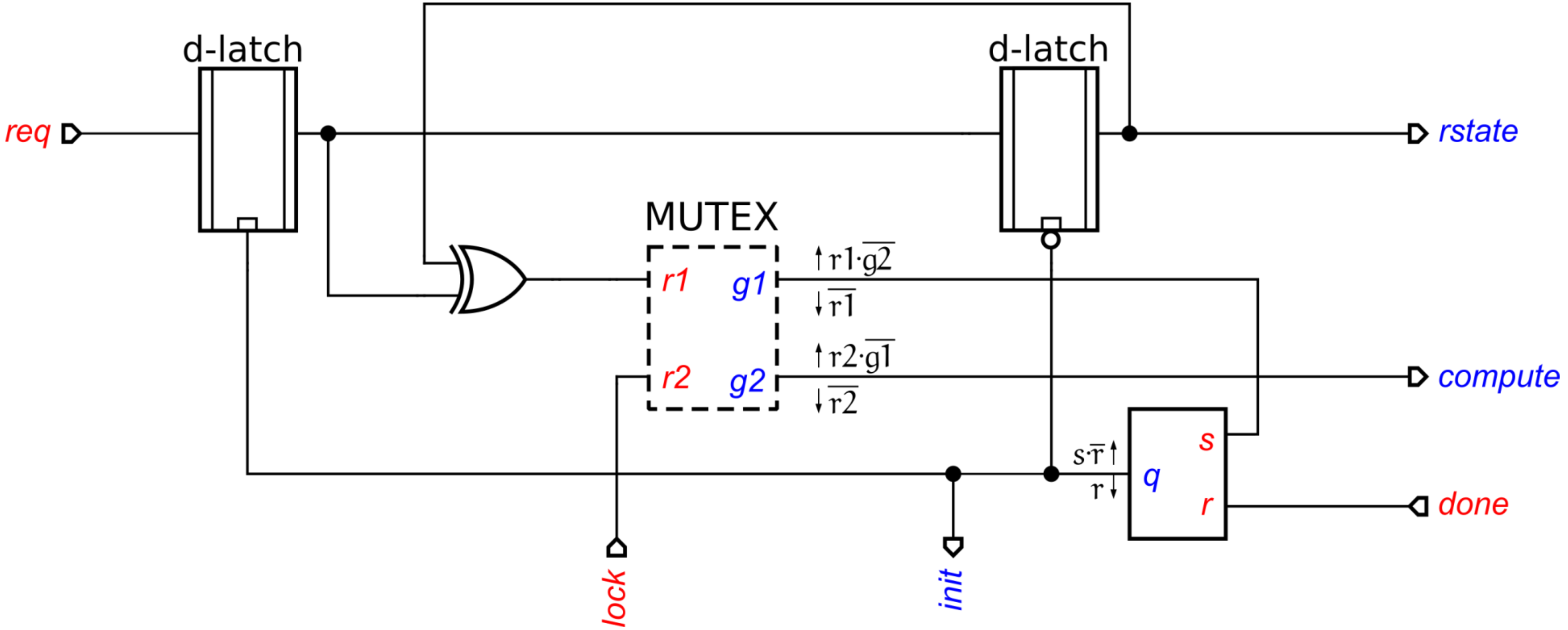


Top-level structure

Input channels

The **request interface** of the arbiter:

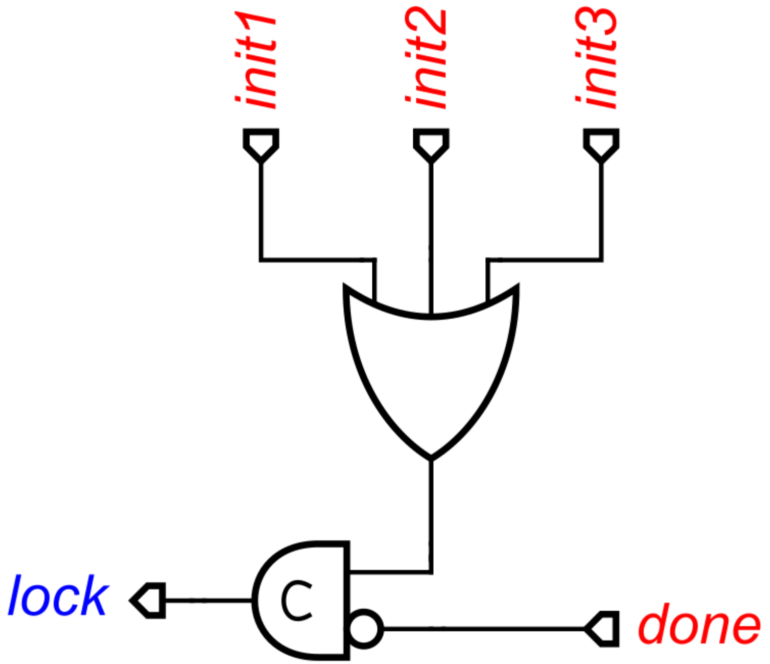
- Accepts arbitration request changes
- Activates the Lock controller to start the new arbitration round
- Provides the current request state to the Grant controller

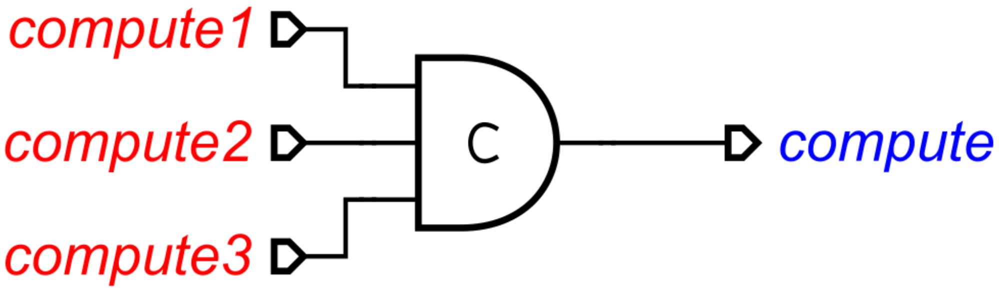


Lock controller

Locks all input channels to create the locked request state

- Initialised by one (or more) input channels
- Activates grant controller when all request states are ready
- Unlocks all input channels when the grant controller has finished



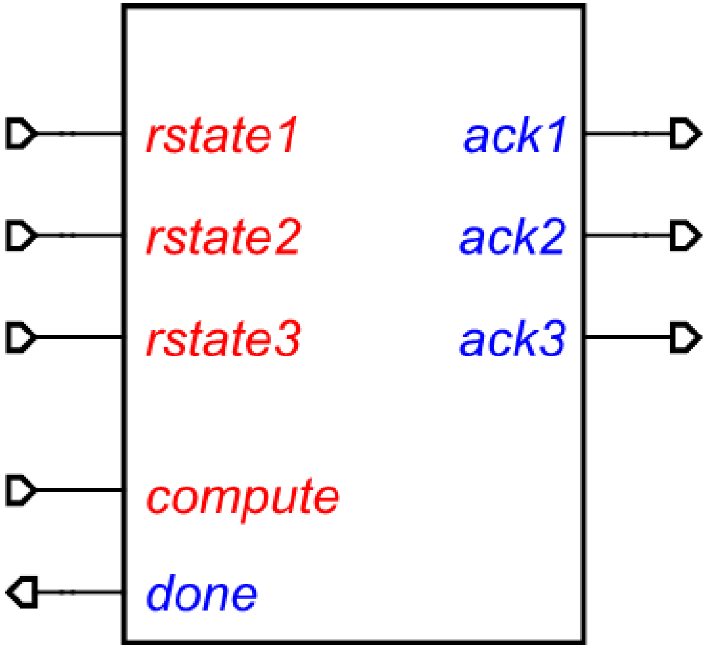


Grant controller

Grant requests subject to constraints

- Initialised by one (or more) input channels
- Activated when the state of all requests is ready
- Unlocks all input channels once the decision has been made

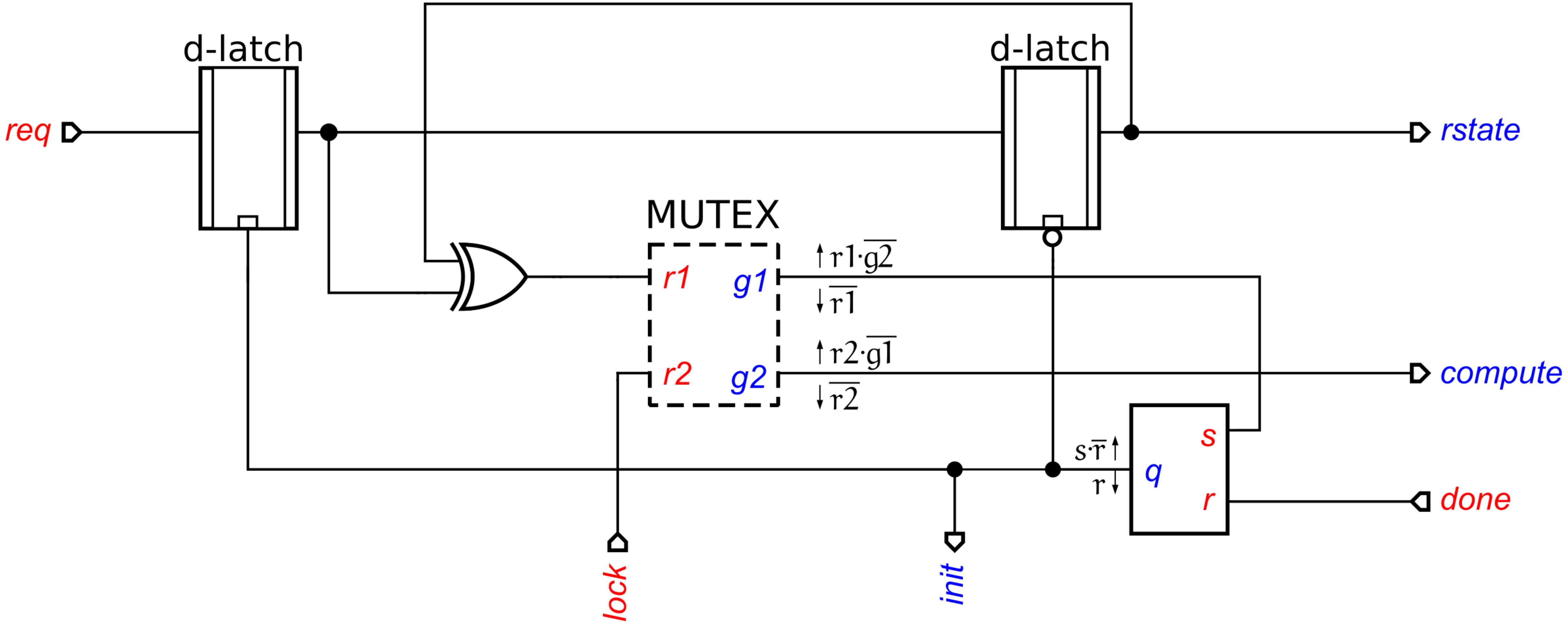
GRANT CONTROLLER



Mutex elements in action

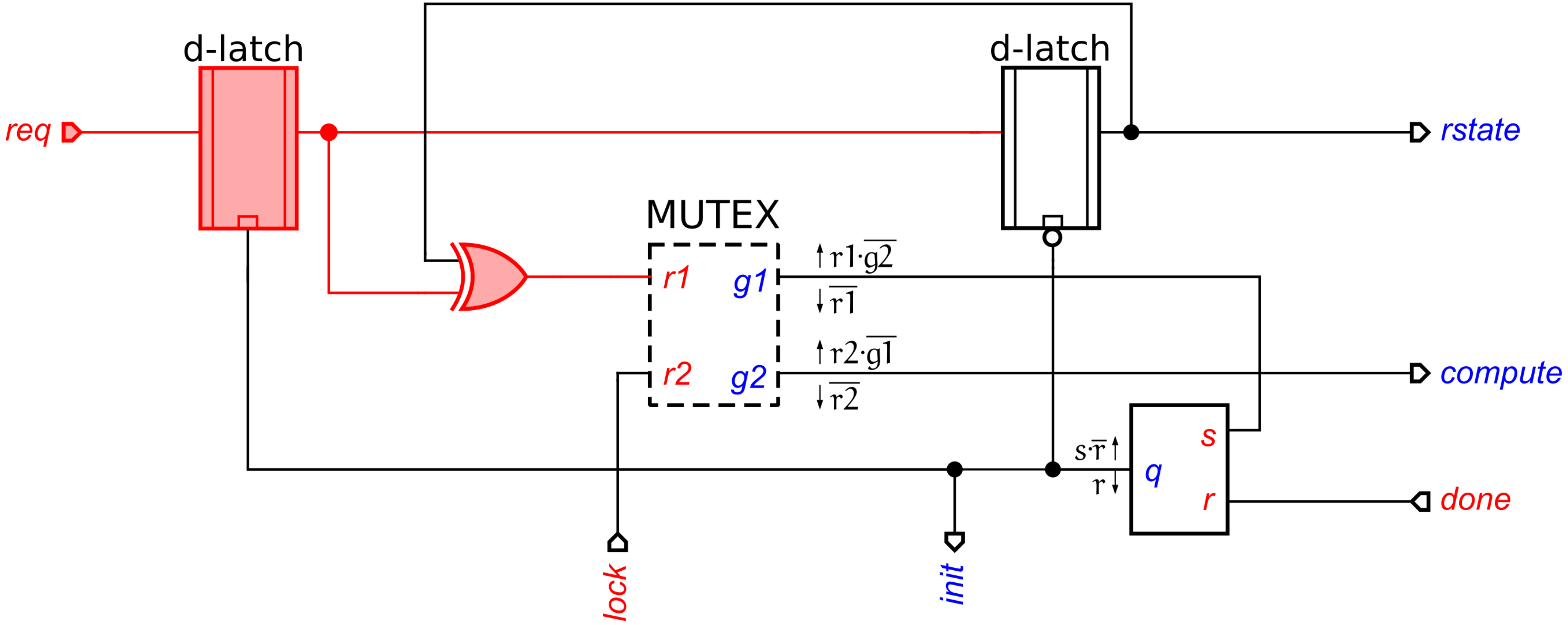
Input Channel in Action (Case 1)

Initial state



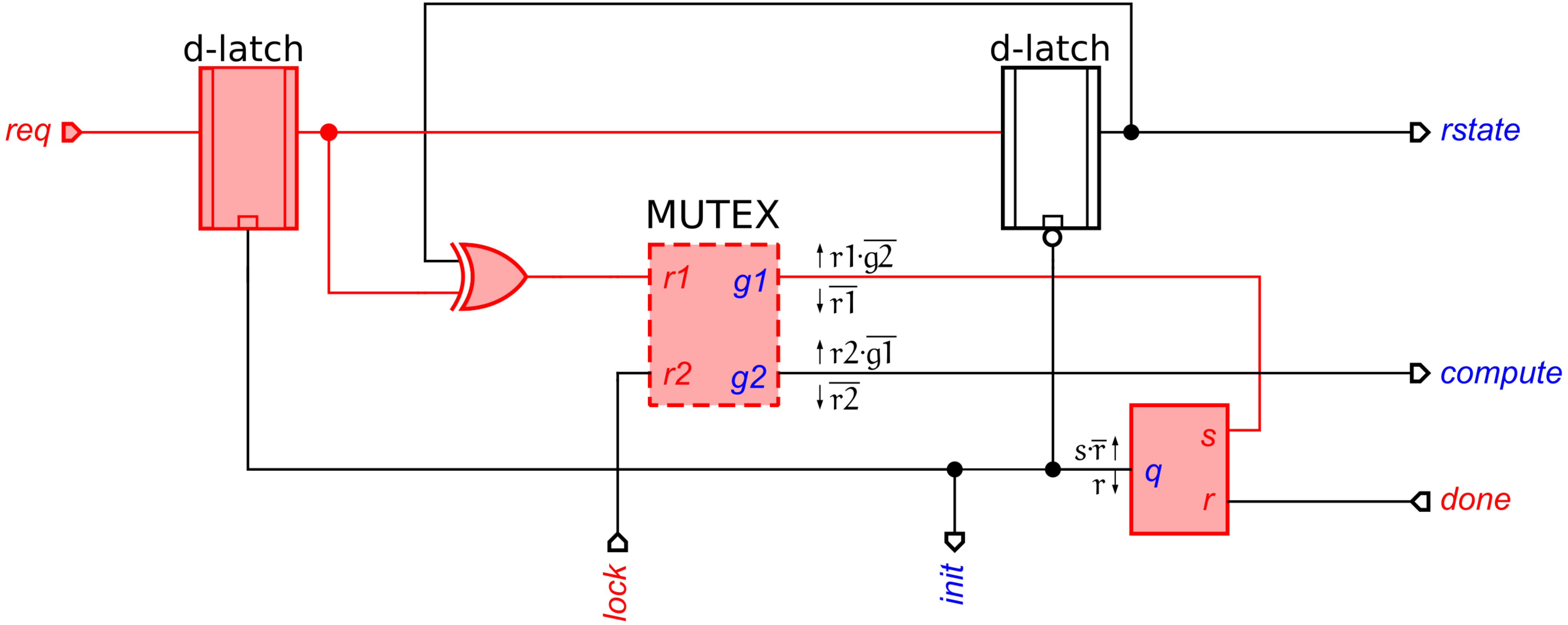
Input Channel in Action (Case 1)

XOR element registers the change of input request



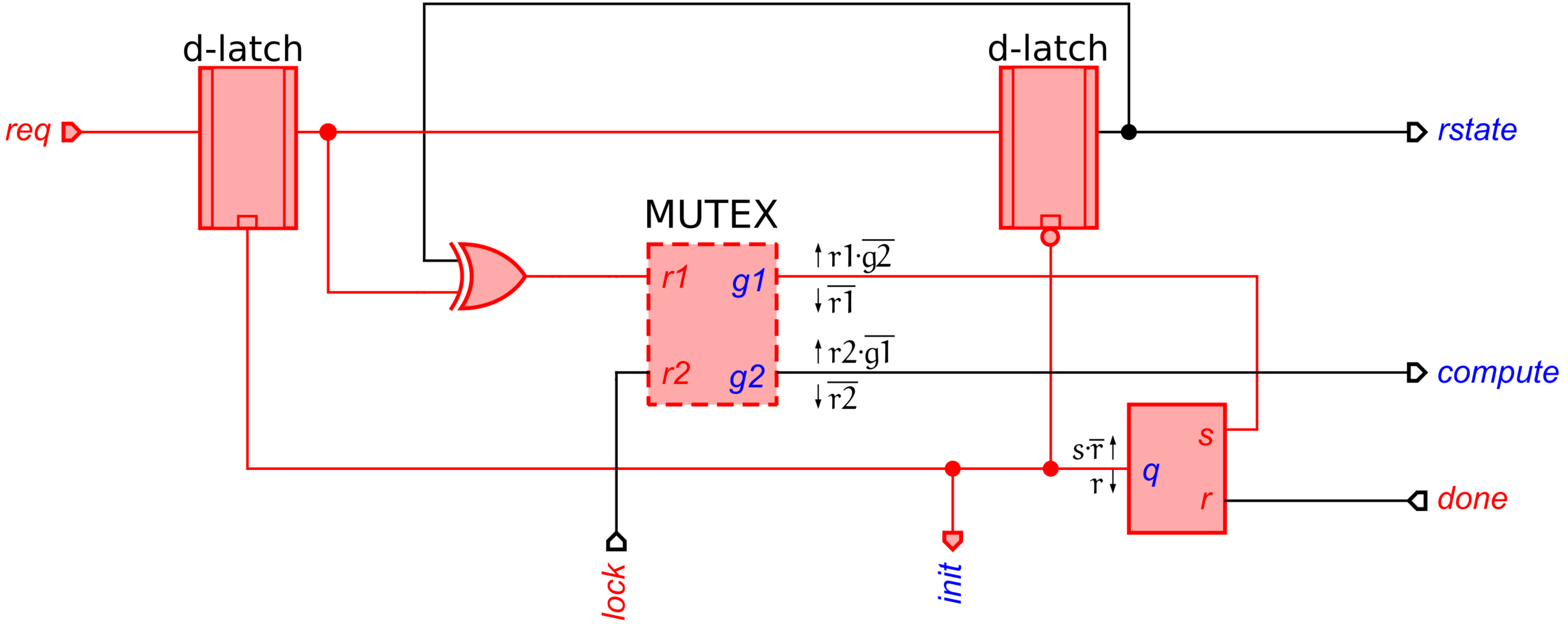
Input Channel in Action (Case 1)

Request change has won the arbitration



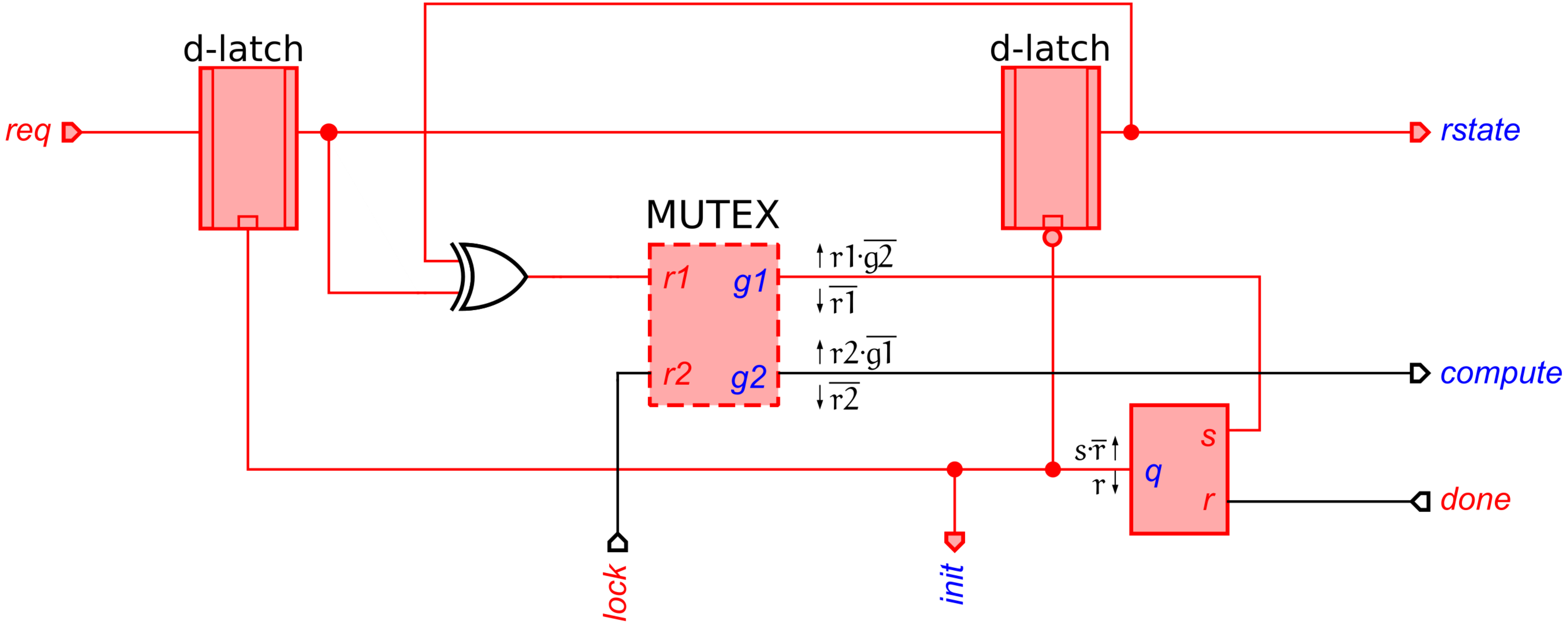
Input Channel in Action (Case 1)

Initialise the lock controller with **init**



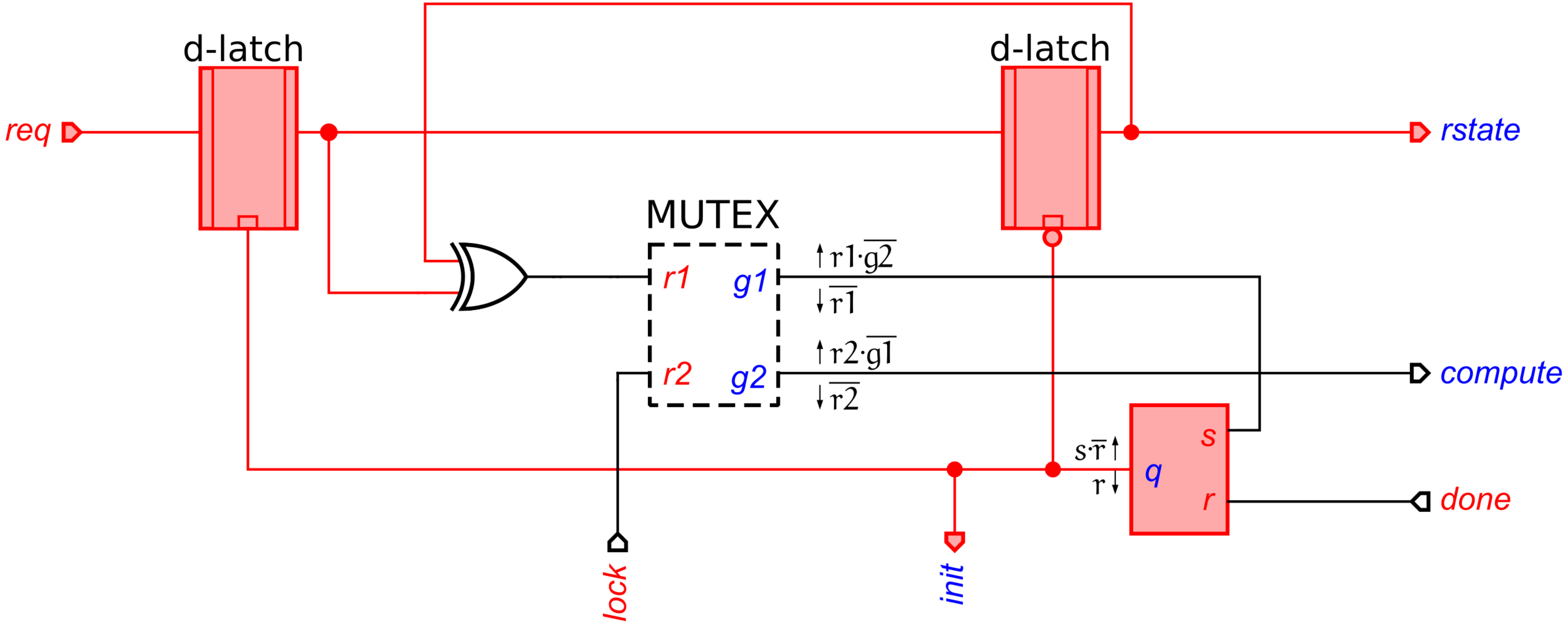
Input Channel in Action (Case 1)

The second d-latch propagates request state



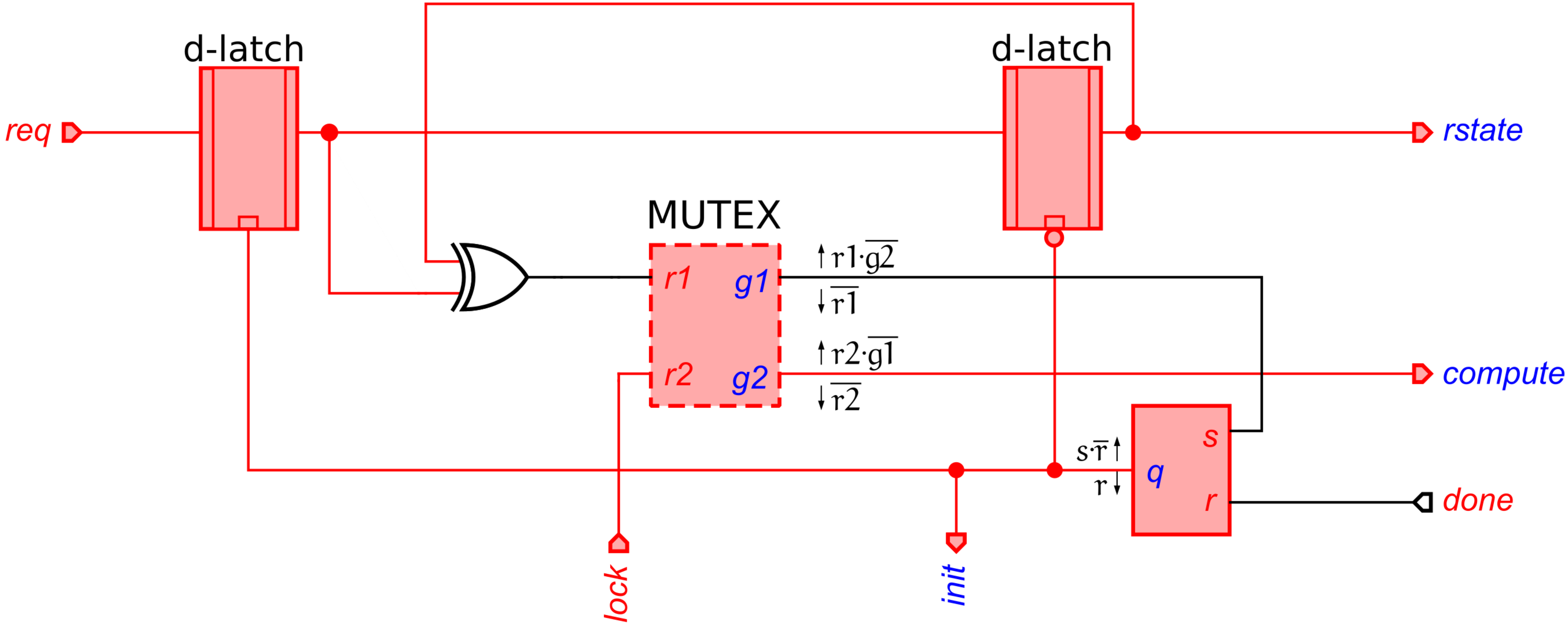
Input Channel in Action (Case 1)

MUTEX is now ready to accept lock signal, rstate is stable



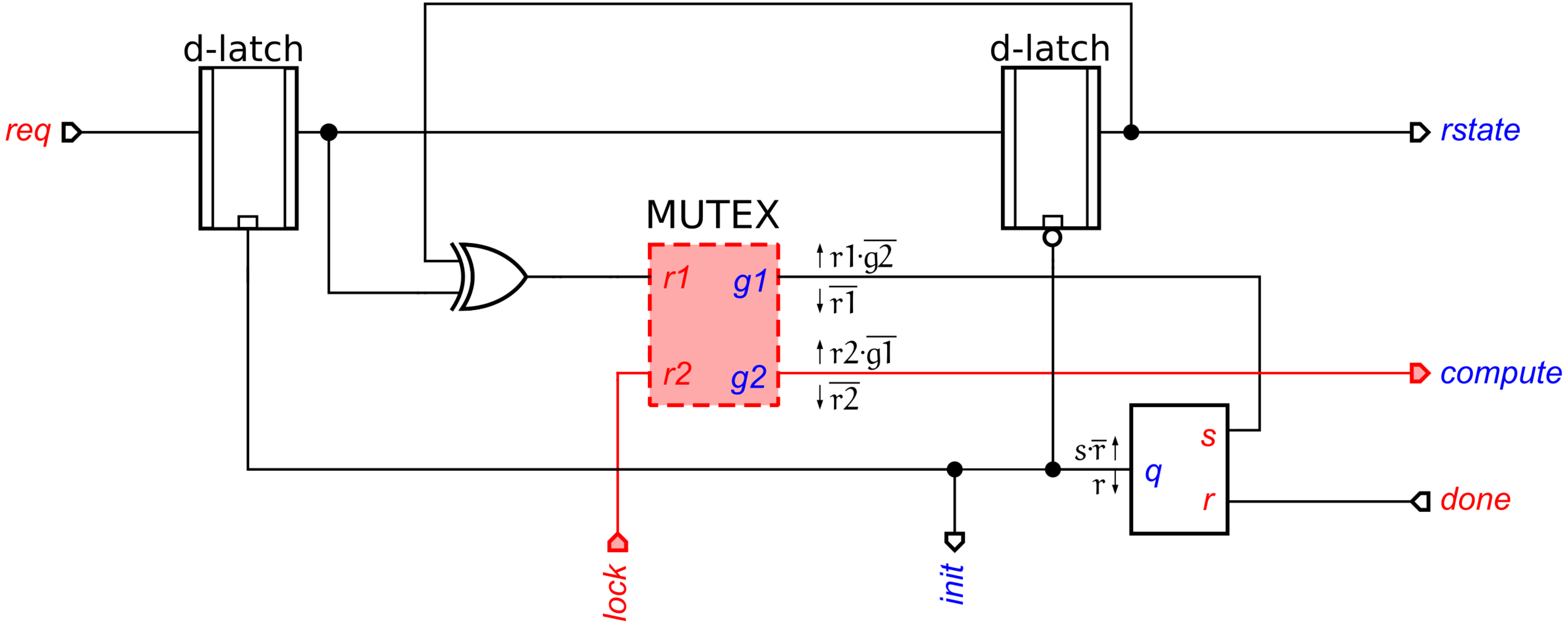
Input Channel in Action (Case 1)

MUTEX accepts lock, initialises ready to compute



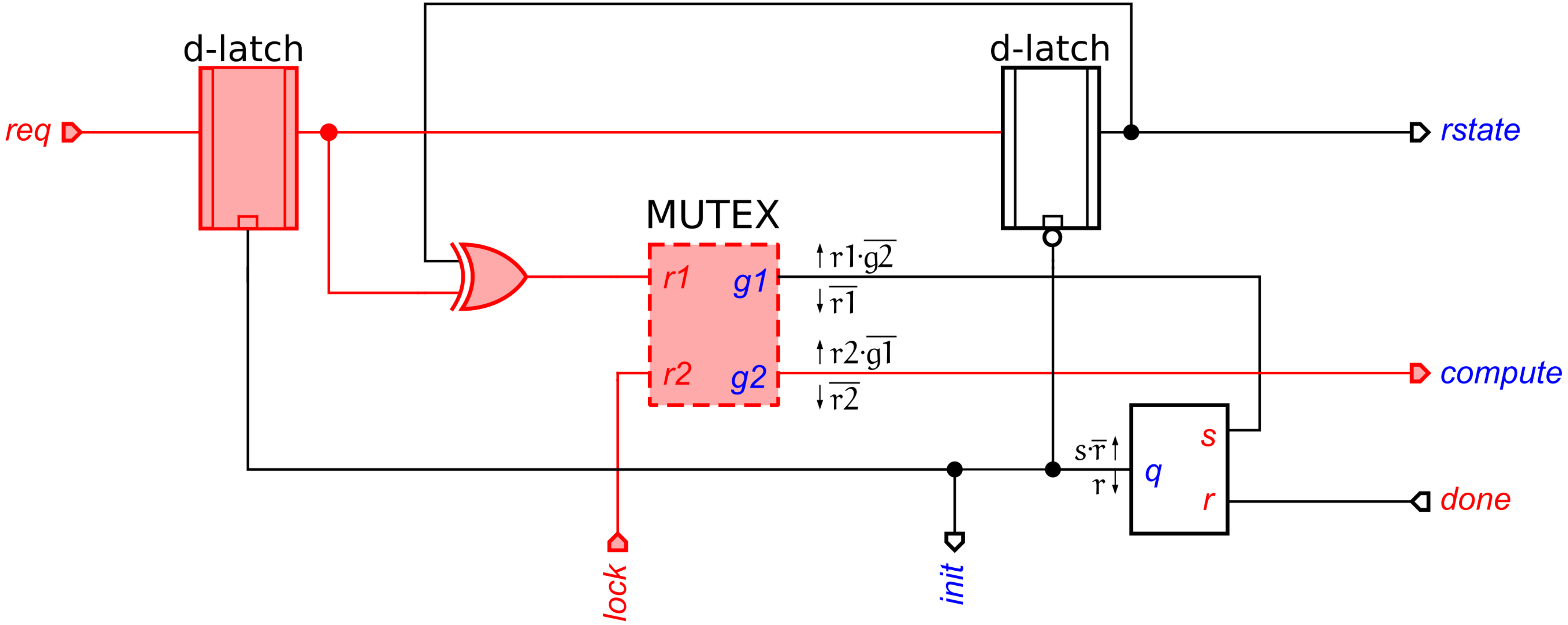
Input Channel in Action (Case 2)

Some other channel has initialised the arbitration



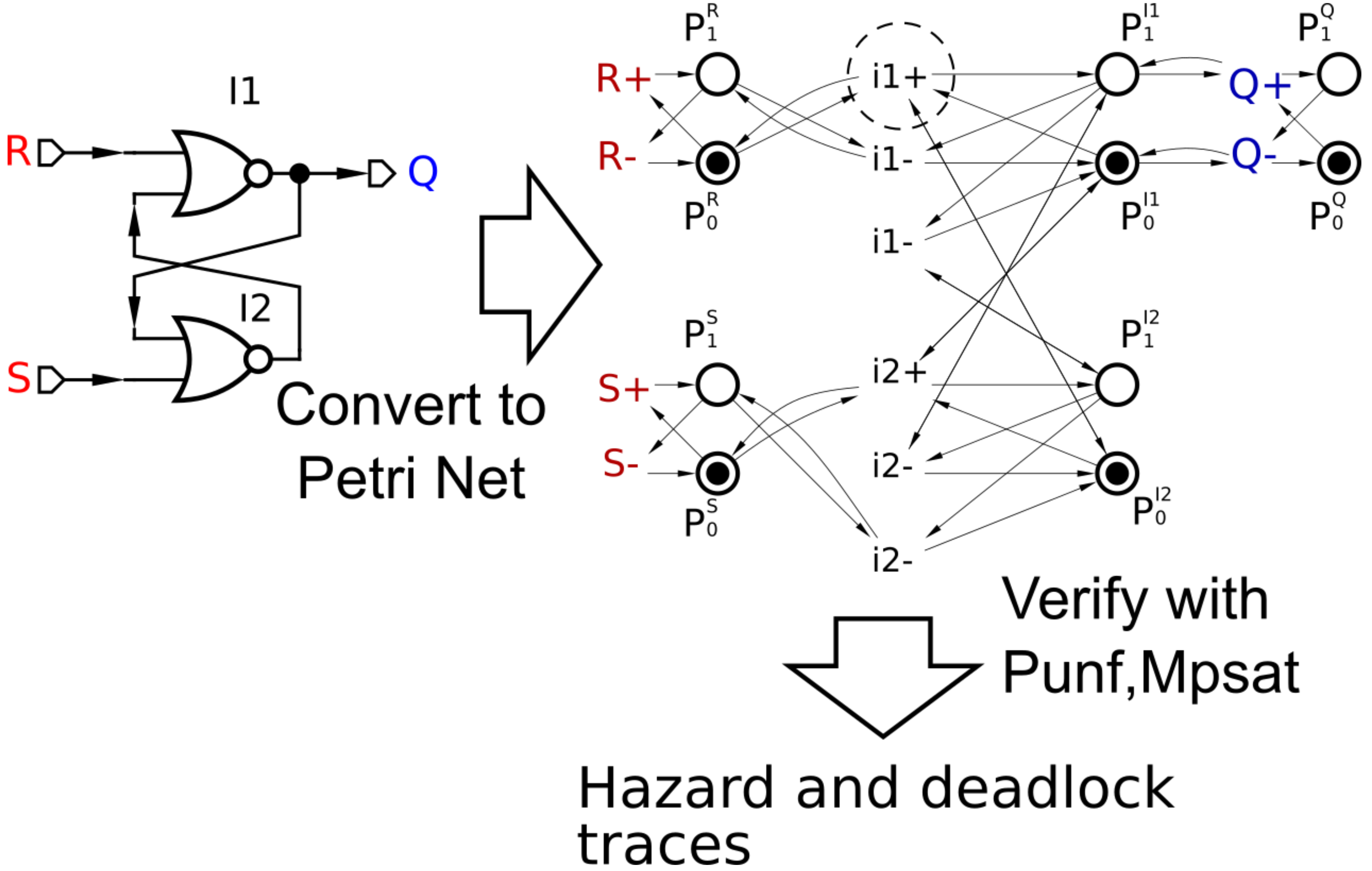
Input Channel in Action (Case 2)

Any req changes will not affect rstate until the end of computation



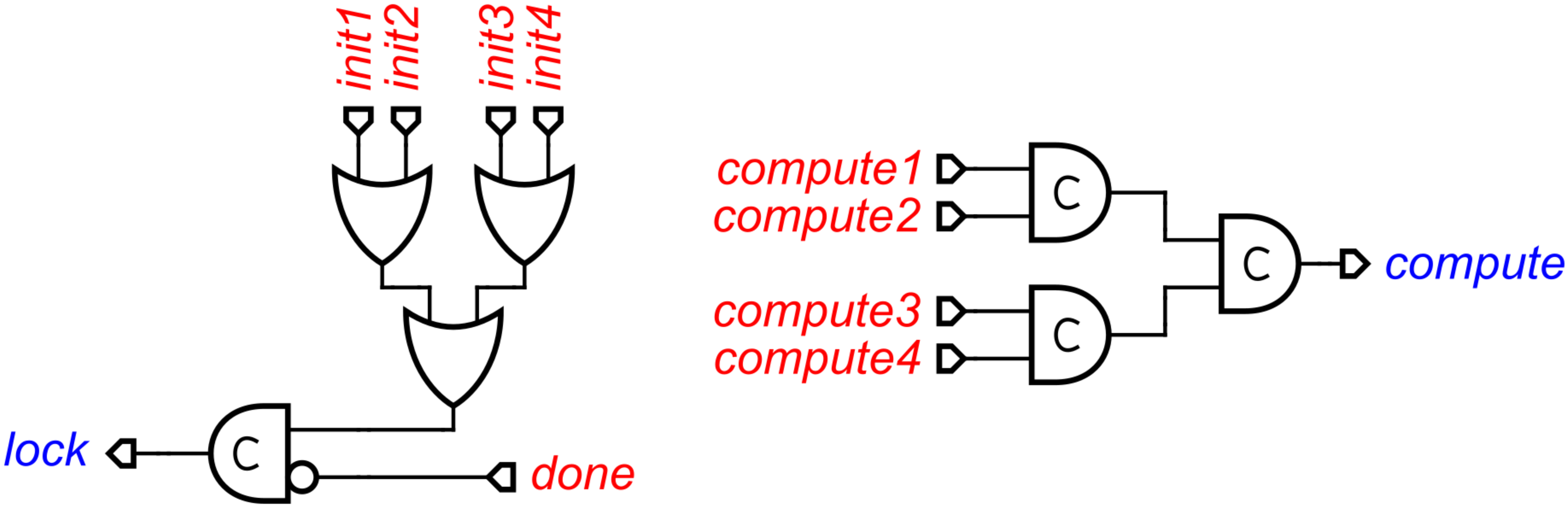
Design issues

Verification flow



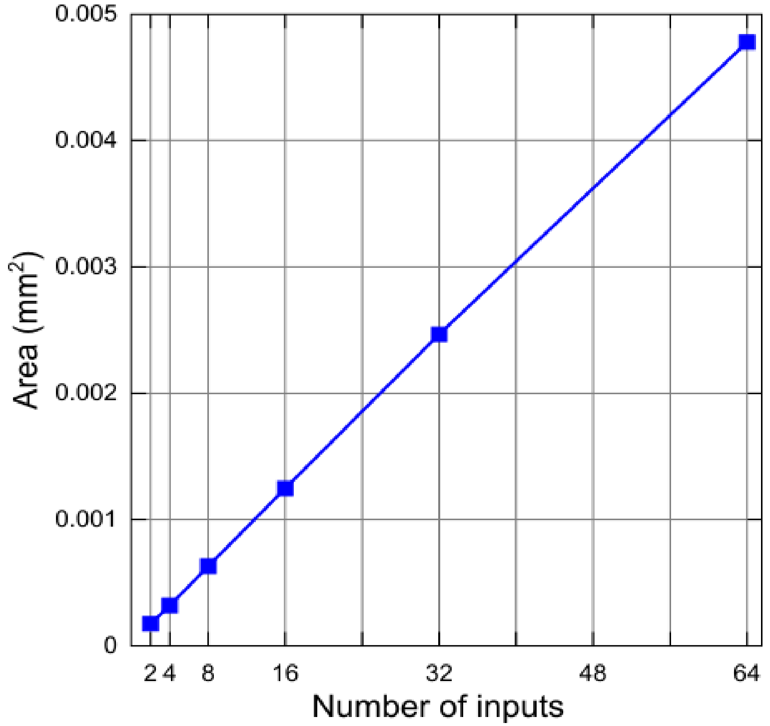
Scaling the Lock Controller

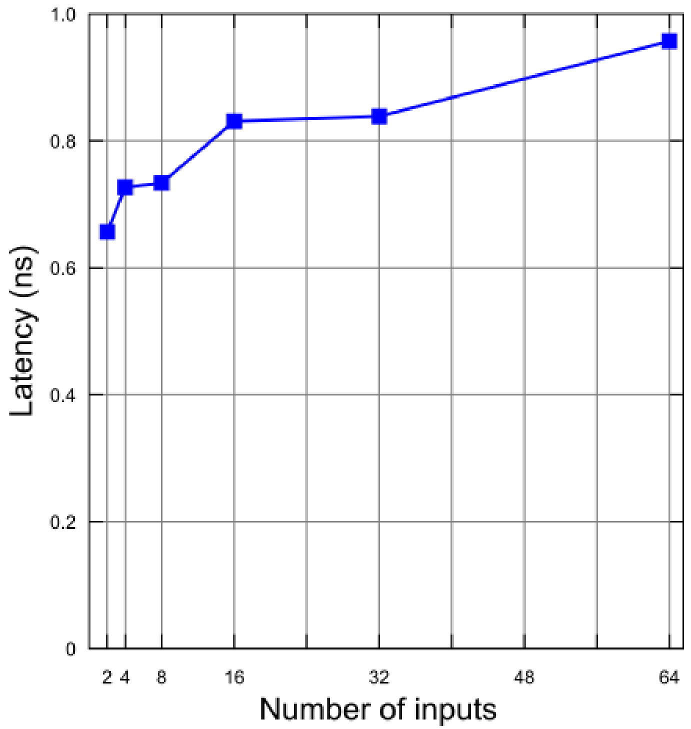
Timing assumption: all OR-gates should settle faster than the delay of the C-element tree and the grant controller

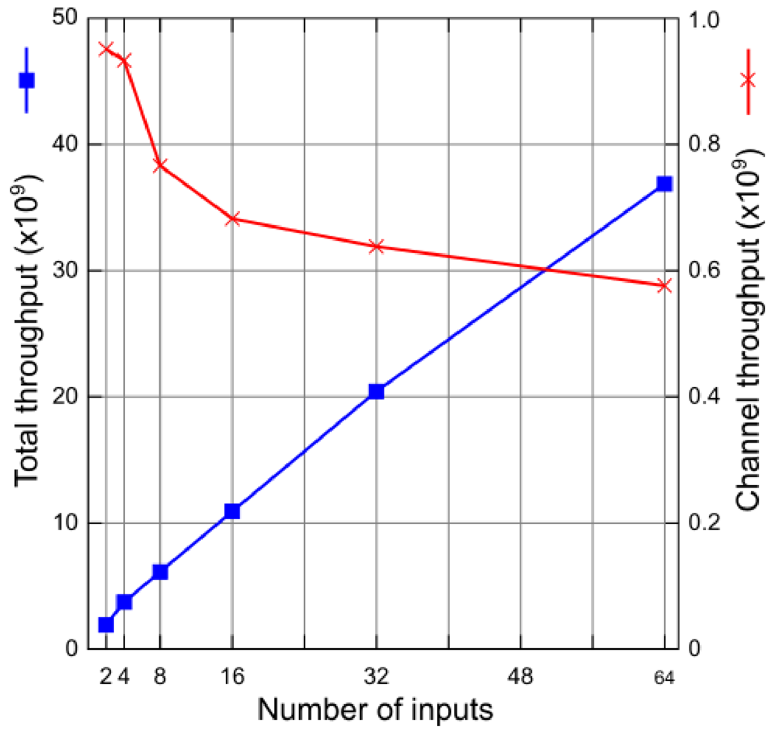


Performance

Design scales linearly with the increased number of input channels
Latency scales logarithmically, because of the tree structures

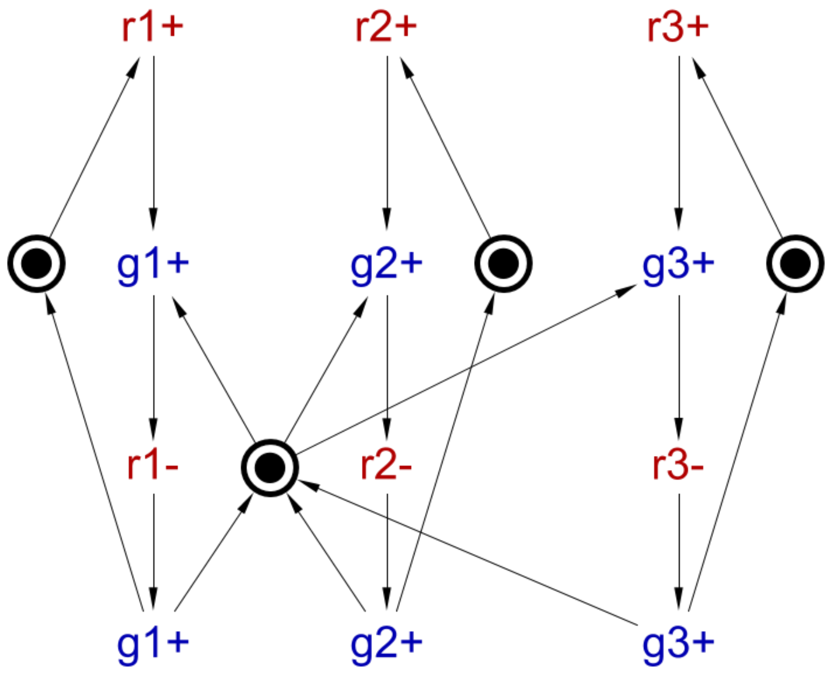






Example: 1-of-3 Arbiter

Arbitration: 1 resource is shared among 3 users.

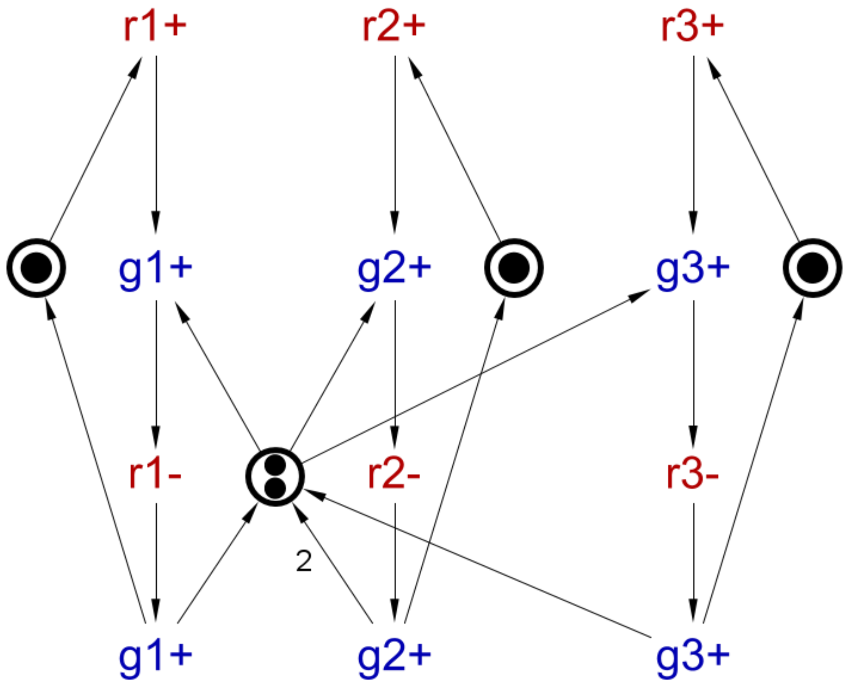


$r1$	$g1$	$r2$	$g2$	$r3$	$g3$	$g1'$	$g2'$	$g3'$
1	X	X	0	X	0	1		
1	X	X	X	0	1	1		
1	X	0	X	X	0	1		
X	X	1	1	X	X		1	
0	X	1	X	0	X		1	
0	X	1	X	X	0		1	
X	X	X	X	1	1			1
0	X	0	X	1	X			1
other combinations						0	0	0

$$g1' = r1 \cdot (\overline{g3} \cdot (\overline{g2} + \overline{r2}) + \overline{r3} \cdot g3)$$
$$g2' = r2 \cdot (\overline{r1} \cdot (\overline{g3} + \overline{r3}) + g2)$$
$$g3' = r3 \cdot (\overline{r1} \cdot \overline{r2} + g3)$$

Example: 2-of-3 Arbiter

Arbitration: 2 resources are shared among 3 users.



$r1$	$g1$	$r2$	$g2$	$r3$	$g3$	$g1'$
1	X	0	X	X	X	1
1	X	X	0	X	X	1
1	X	X	X	0	X	1
1	X	X	X	X	0	1
other combinations						0

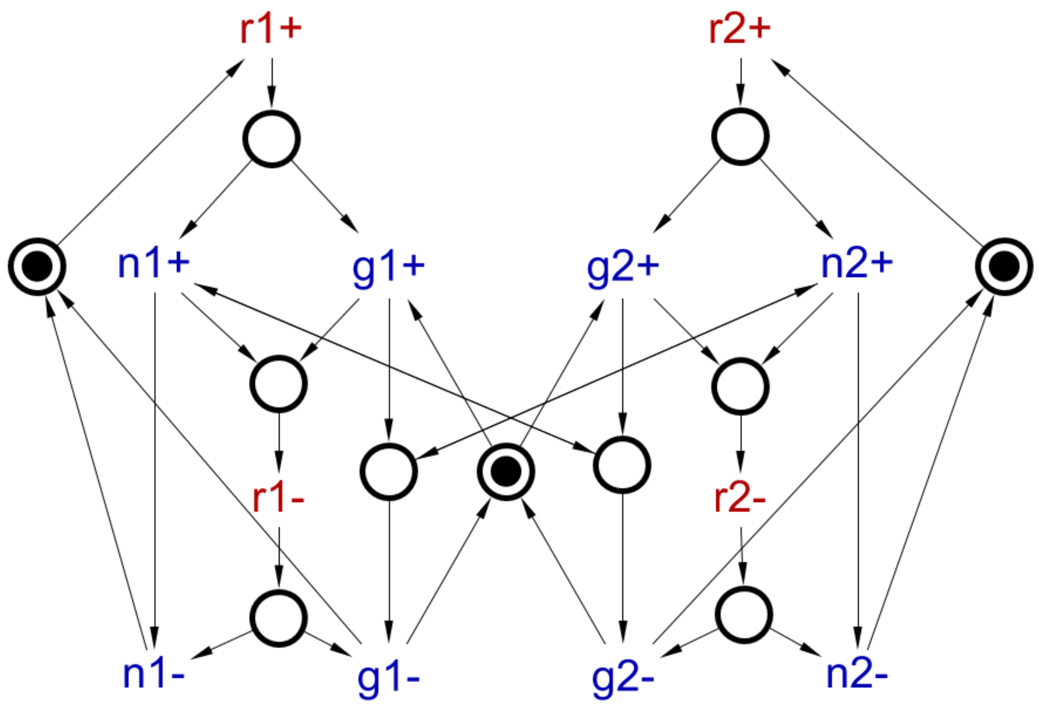
$r1$	$g1$	$r2$	$g2$	$r3$	$g3$	$g2'$
0	X	1	X	X	X	1
X	X	1	1	X	X	1
X	X	1	X	0	X	1
X	X	1	X	X	0	1
other combinations						0

$r1$	$g1$	$r2$	$g2$	$r3$	$g3$	$g3'$
0	X	X	X	1	X	1
X	X	0	X	1	X	1
X	X	X	X	1	1	1
other combinations						0

$$\begin{aligned} g1' &= r1 \cdot \overline{r2 \cdot g2 \cdot r3 \cdot g3} \\ g2' &= r2 \cdot (\overline{r1 \cdot g3 \cdot r3} + g2) \\ g3' &= r3 \cdot (\overline{r1 \cdot r2} + g3) \end{aligned}$$

Example: Nacking Arbiter

Nacking actively acknowledges that the resource is occupied, the user can attempt to take some other action rather than waiting for grant



$r1$	$g1$	$n1$	$r2$	$g2$	$n2$	$g1'$	$n1'$	$g2'$	$n2'$
1	X	X	0	X	X	1			
1	X	X	X	0	X	1			
1	X	X	1	1	X		1		
0	X	X	1	X	X			1	
X	X	X	1	1	X			1	
1	X	X	1	0	X				1
other combinations						0	0	0	0

$$
\begin{aligned}
g1' &= r1 \cdot \overline{r2 \cdot g2 \cdot r3 \cdot g3} \\
g2' &= r2 \cdot (\overline{r1 \cdot g3 \cdot r3} + g2) \\
g3' &= r3 \cdot (\overline{r1 \cdot r2} + g3)
\end{aligned}
$$

Main results

Generalised asynchronous arbiter:

- Low latency, deadlock-free, arbitrary decision logic
- Clean top-level decomposition: Input Channels, Lock and Grant
- Same general structure can be used to design different arbiters
- Grant controller can be synthesised automatically

Problem solved?

- In practical terms, the answer is **“Probably Yes”**
- If you are a theoretician then the answer is **“No”**: still no synthesis!