Deadlock Verification in Network-on-Chips

Julien Schmaltz and Freek Verbeek

Radboud University Nijmegen

Open Universiteit
www.ou.nl

Julien and Freek

- Julien
  - French
  - Ph.D. from University of Grenoble 2006 (D. Borrione, TIMA Labs)
  - 1-year postdoc in Saabrücken, Germany
  - 2,5-year postdoc Radboud University Nijmegen
  - Since October 2009, Assistant Professor Open Universiteit
    - but I still have an office within RUN !
  - Main research area in formal methods

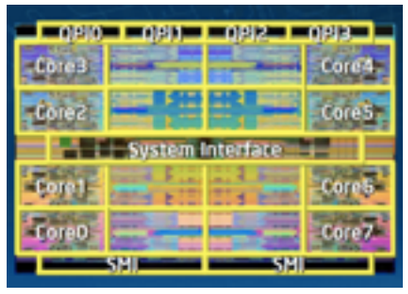
- Freek
  - Dutch
  - Ph.D. student at the Radboud University Nijmegen
  - Started October 2008
  - Main research area in deadlock verification for NoCs
  - Official promotors: Frits Vaandrager and Marko van Eekelen
  - Daily supervisor: Julien

Mutlicore shift has happened (A. Agarwal - Keynote NOCS 2011)

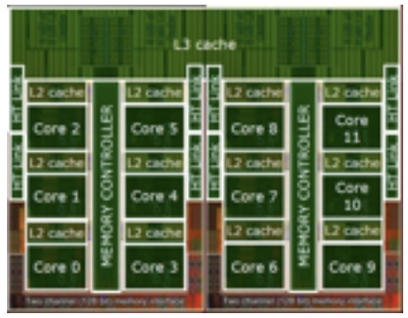
www.ou.nl

Growing number of cores (W. Tichy - Keynote ICST 2011)

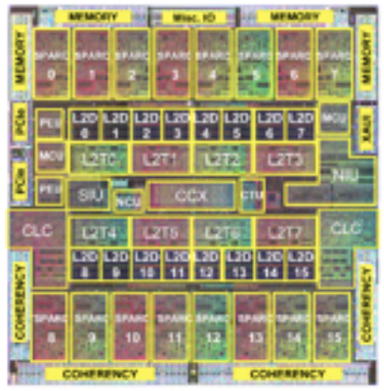
Intel 8 cores
~2.3 Bill. T. on 6.8cm^2



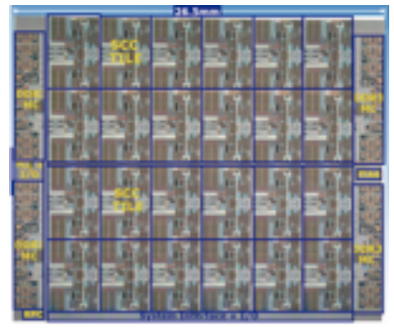
AMD Opteron 12 cores
~1.8 Bill. T. on 2x3.46cm^2



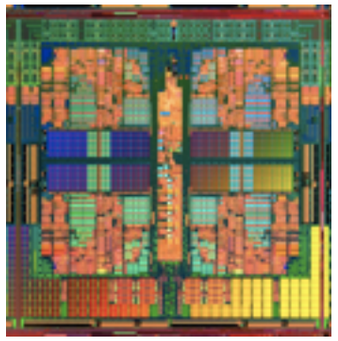
Sun Niagara3 16 cores
~1 Bill. T. on 3.7cm^2



Intel SCC 48 cores
~1.3 Bill. T. on 5.6cm^2

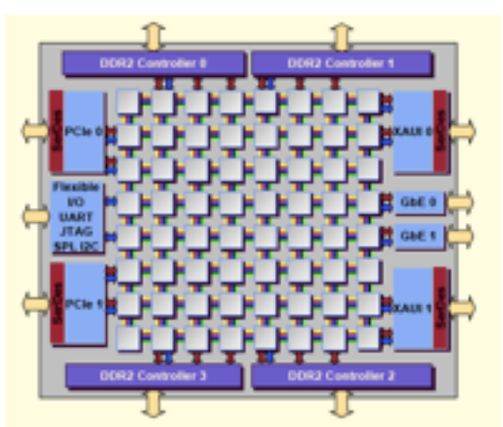


Intel 4 cores
~582 Mio. T. on 2.86cm^2

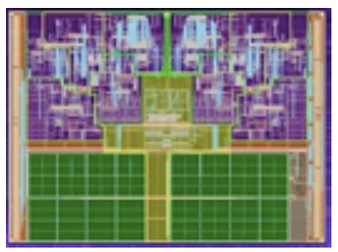


Intel Research 80 cores
~100 Mio. T. on 2.75cm^2

Tilera TILEPro64 64 cores



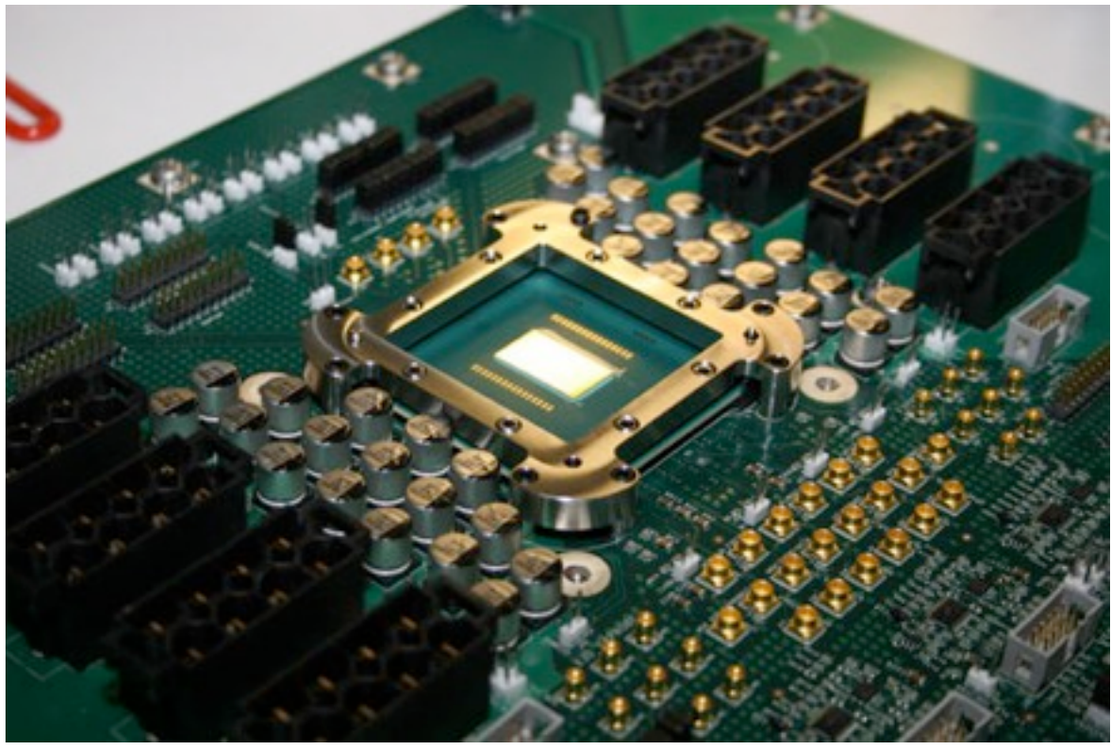
Intel 2 cores
~167 Mio. T. on 1.1cm^2



www.ou.nl

80 Cores Research Chip

- Teraflops, 62 Watts
- 100 millions transistors, 275 mm2
- 25% node area for router



- ASCI Red Supercomputer
- Teraflops (Dec. 1996)
- 10, 000 Pentium Pro
- 104 cabinets, 230 m2

It is only the beginning ...

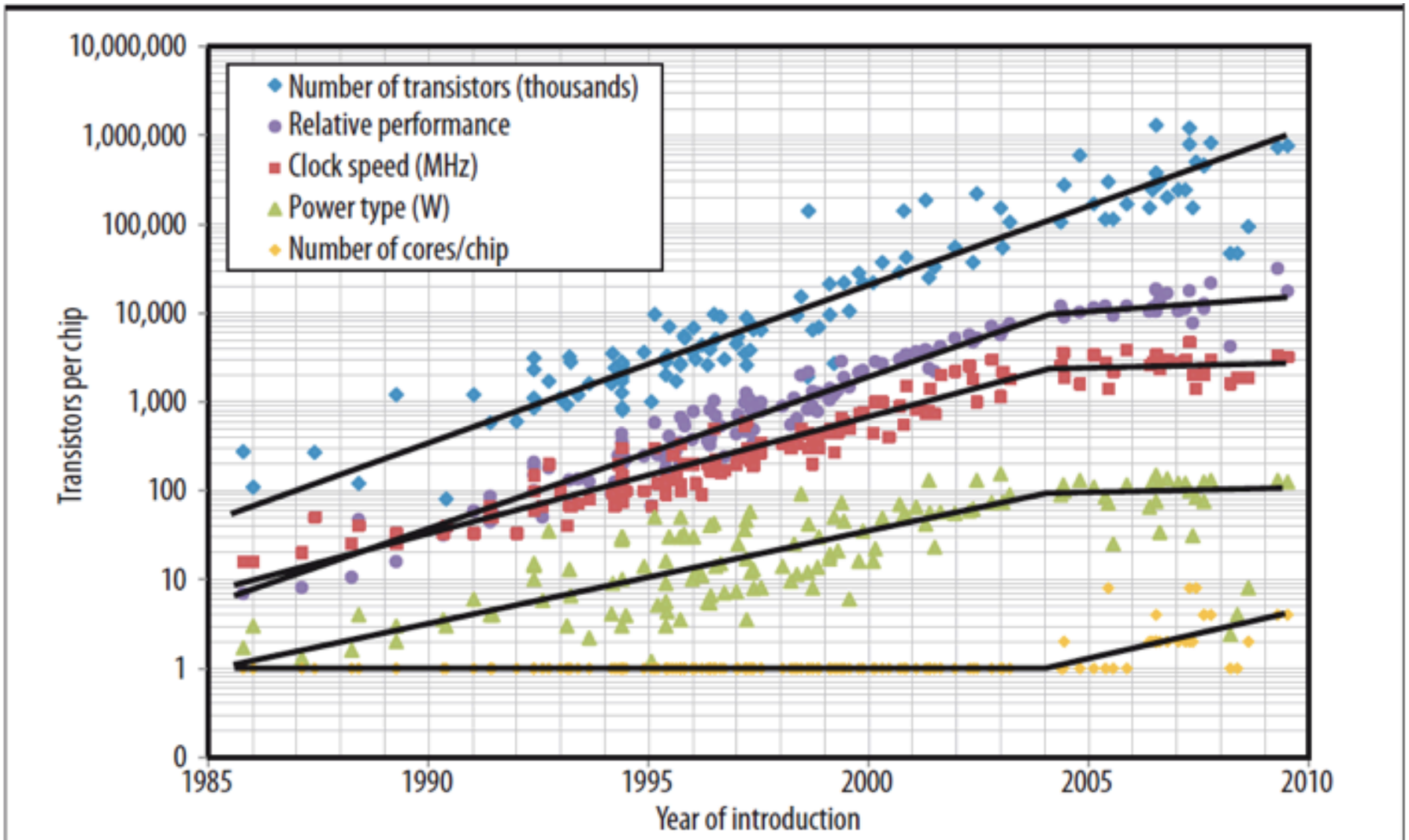


Figure 1. Transistors, frequency, power, performance, and processor cores over time. The original Moore's law projection of increasing transistors per chip remains unabated even as performance has stalled.

Source. *IEEE Computers* 2011

A key component:
the communication fabric or Network-on-Chip (NoC)

- 80 core research chip
  - 25% area for the NoC
  - 30% power consumption for the NoC

- Communication fabrics key
  - to performance and efficiency
  - to functional correctness

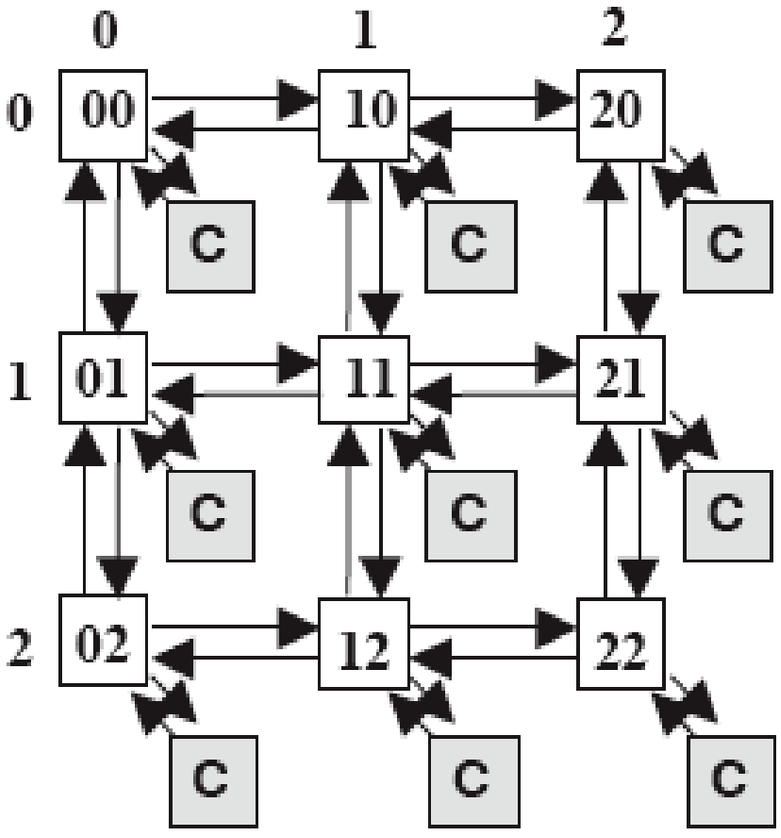
Verification challenges

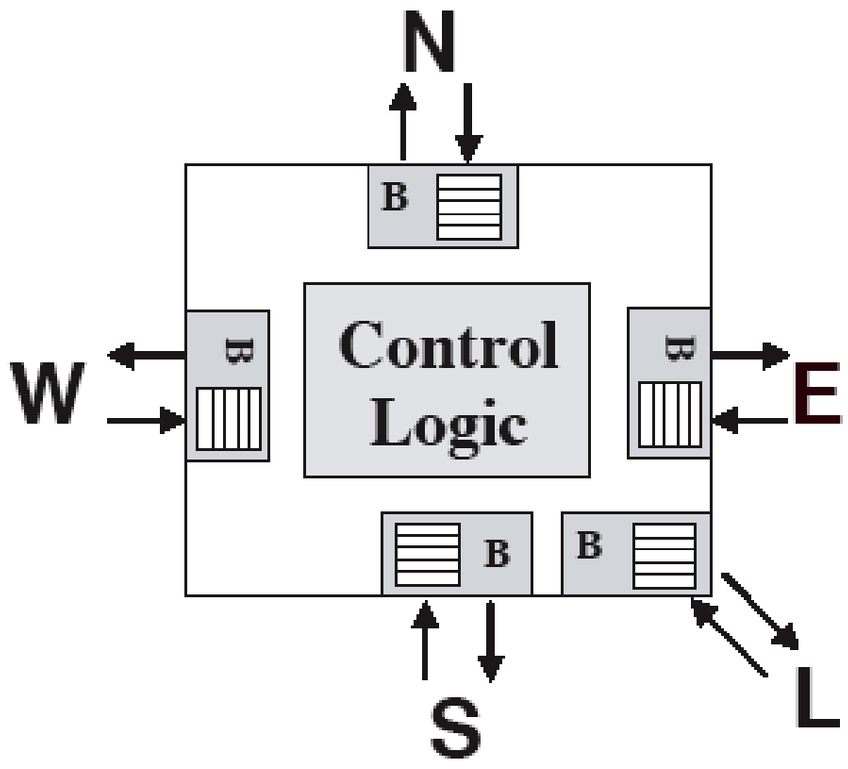
- NoCs are very large systems
  - Verification methods must scale up to 100s of agents
  - Large number of parameters (routing, switching, buffers, etc.)
  - Regular and irregular topologies

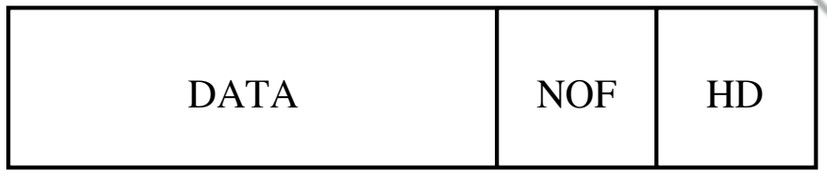
- NoCs must be fault-tolerant
  - Deep sub-micron effect
  - Not all routers/processors are working
  - Static and dynamic fault-models

- NoCs have intricate message dependencies
  - Mix between interconnect and protocols
  - e.g. cache coherency or master/slave
  - Deadlocks can emerge from deadlock-free routing and protocols

Networks-on-Chips: Example 1, Hermes





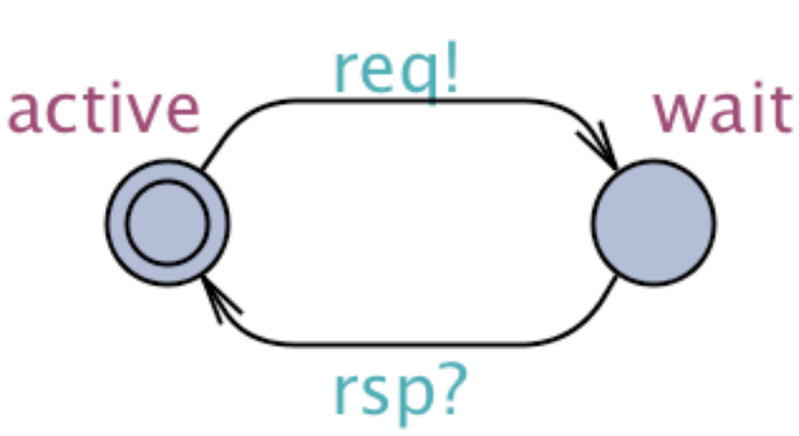


- XY minimal deterministic routing
- Wormhole switching
- Frame structure based on flits (header, control, data)

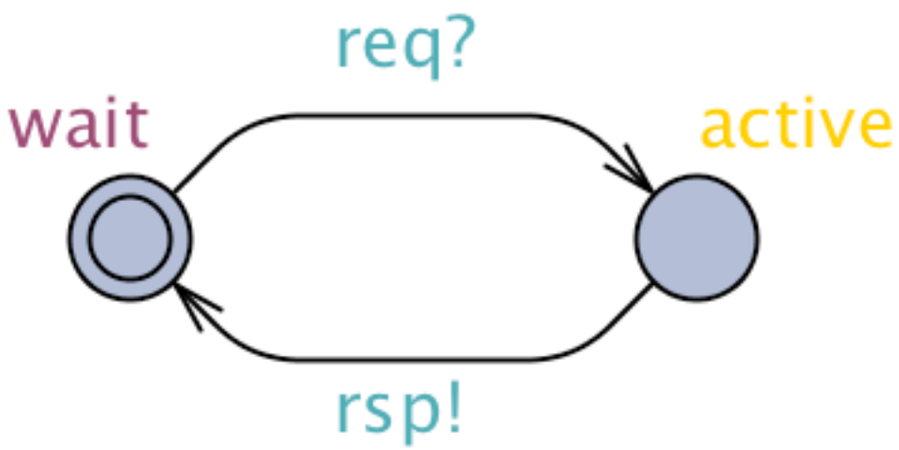
www.ou.nl

A simple master/slave protocol

- Masters send requests and wait for responses
- Slaves produce responses when receiving requests
- Deadlock-free protocol



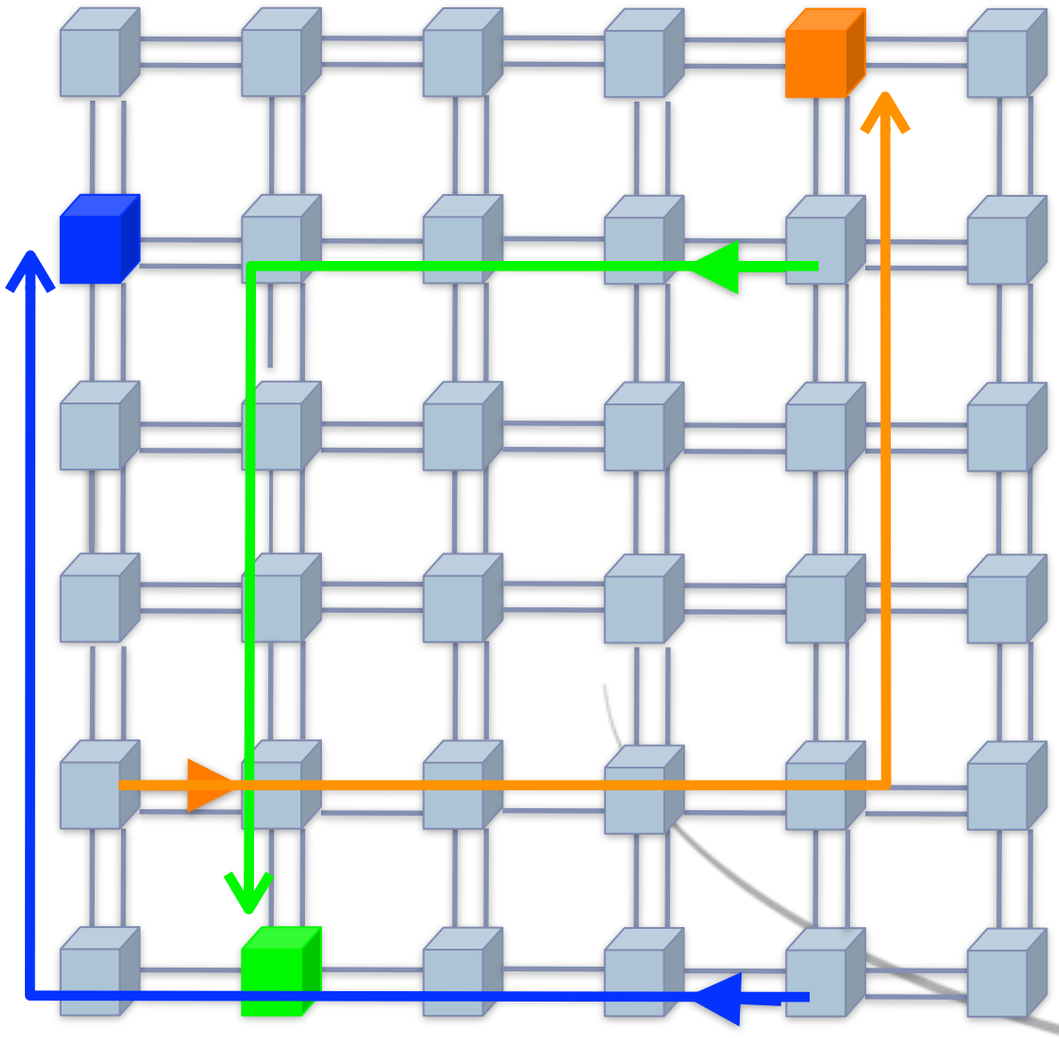
Master



Slave

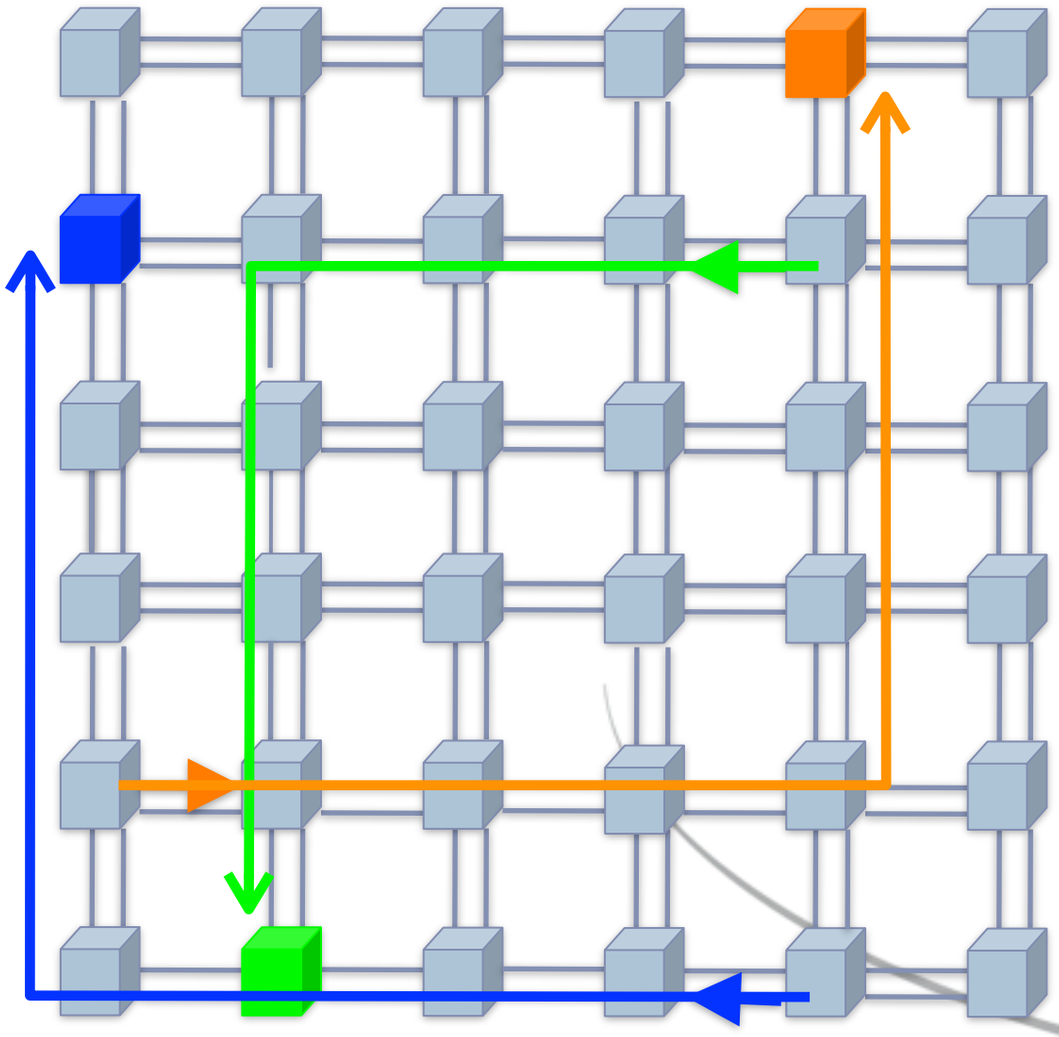
XY routing in a 2D-mesh

- Deterministic simple routing algorithm
- First route to the destination column and then to the correct row
- No cyclic dependencies and thus deadlock-free



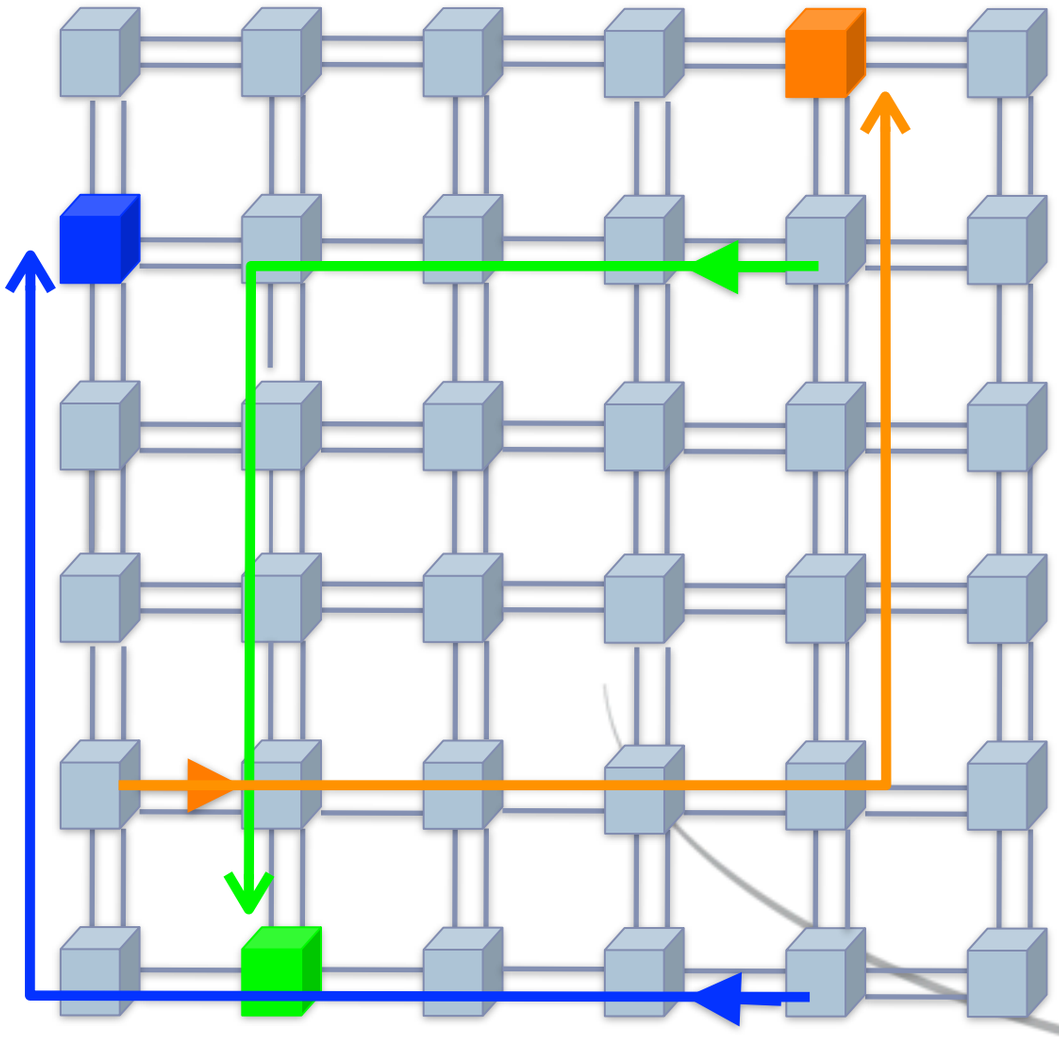
All nodes are both masters and slaves

- Is the system deadlock-free ?



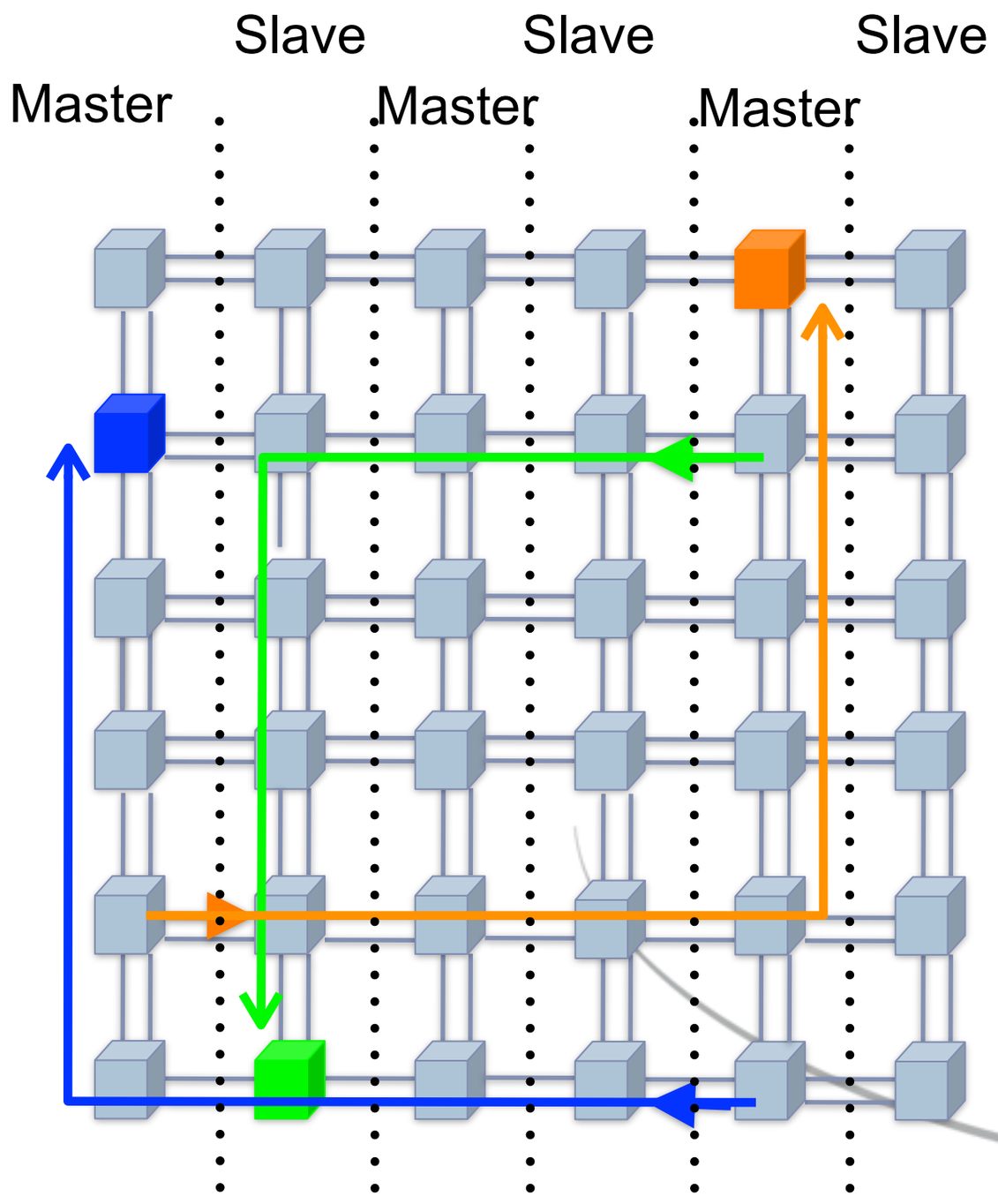
All nodes are both masters and slaves

- Is the system deadlock-free ?
- No ! Two nodes are sufficient to create a deadlock.



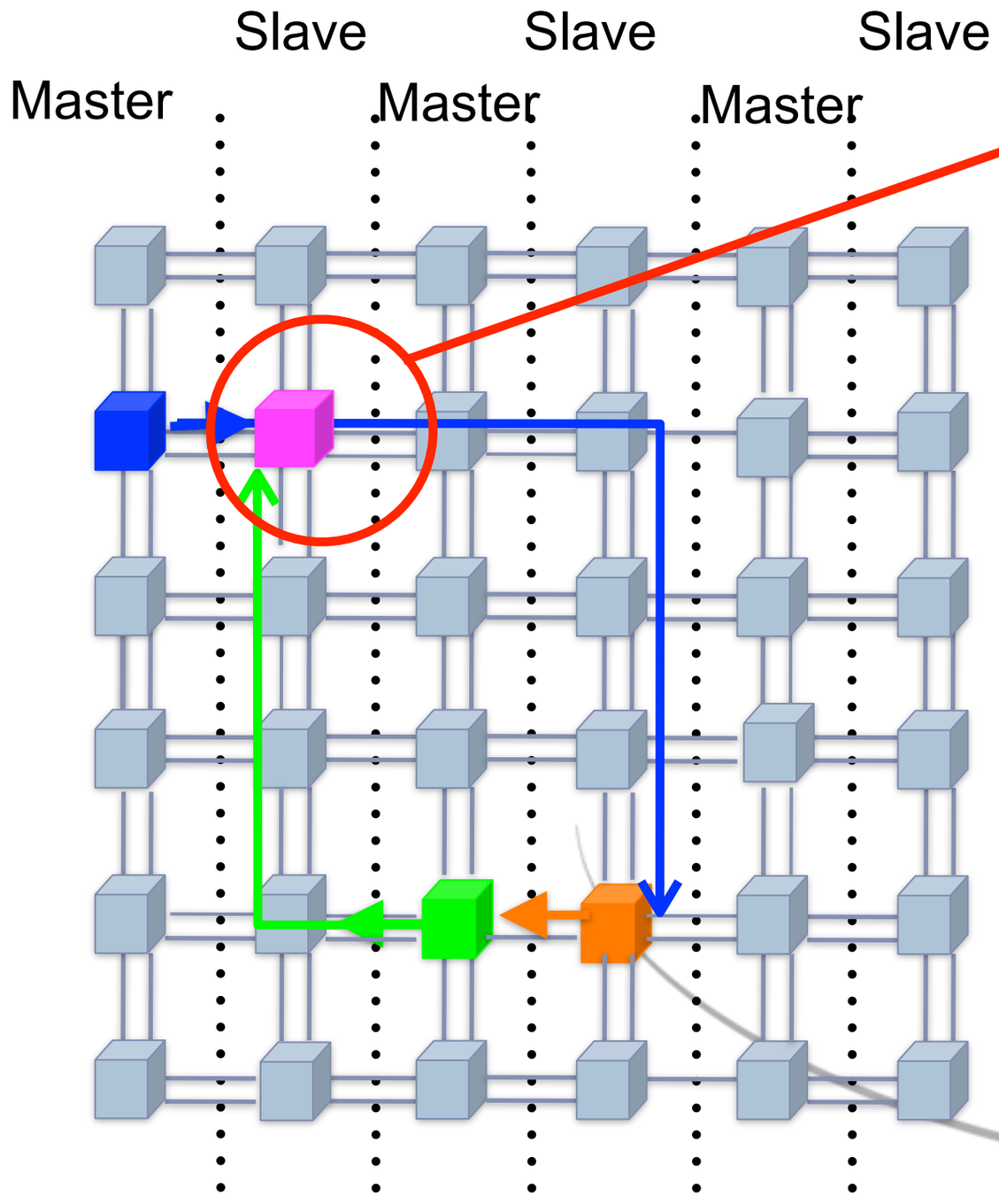
Masters on even columns and slaves on odd columns

- Is the system deadlock-free ?



Masters on even columns and slaves on odd columns

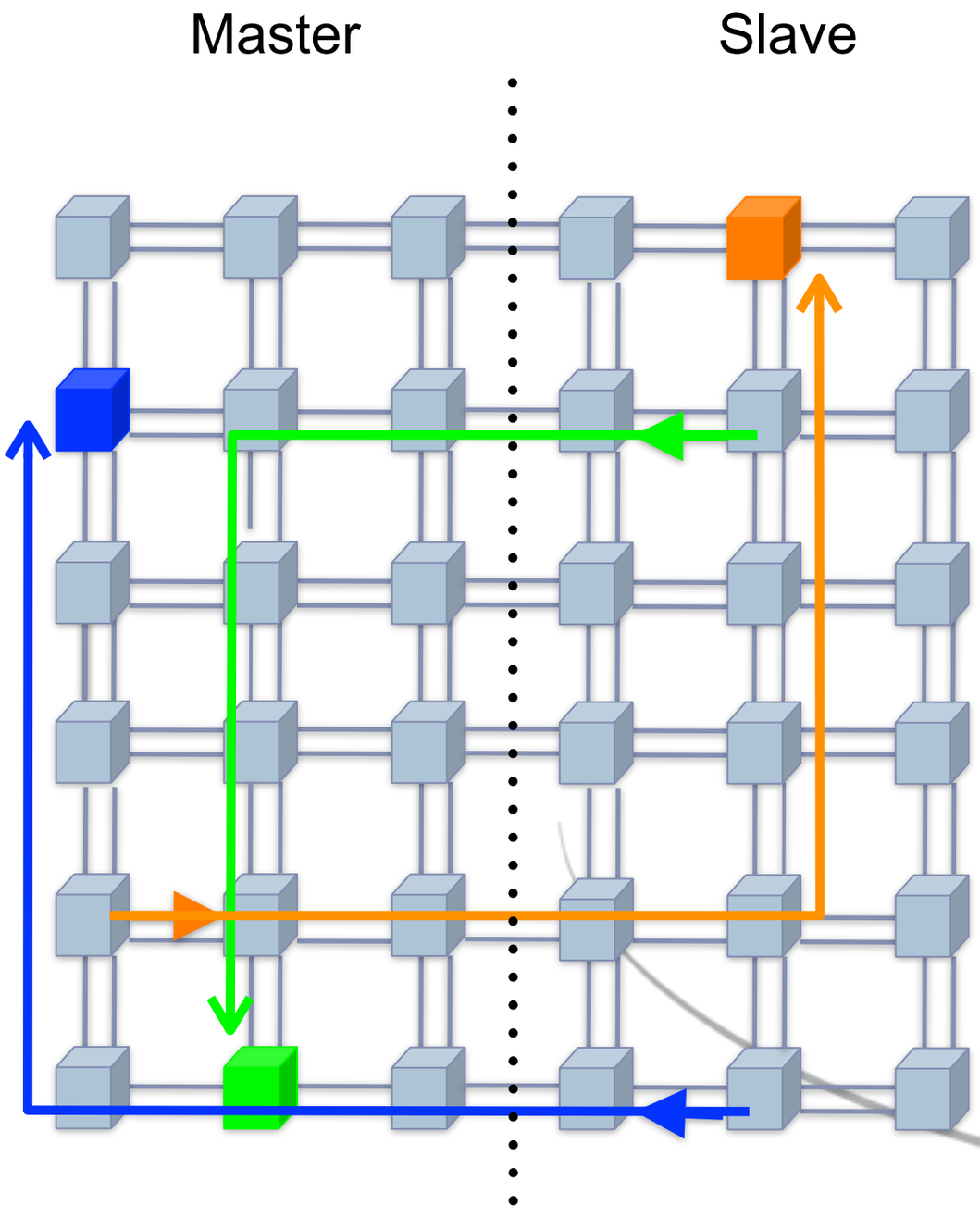
- Is the system deadlock-free ?
- No if at least four columns, yes otherwise.



Green request cannot be consumed as it is waiting for the blue request to leave.

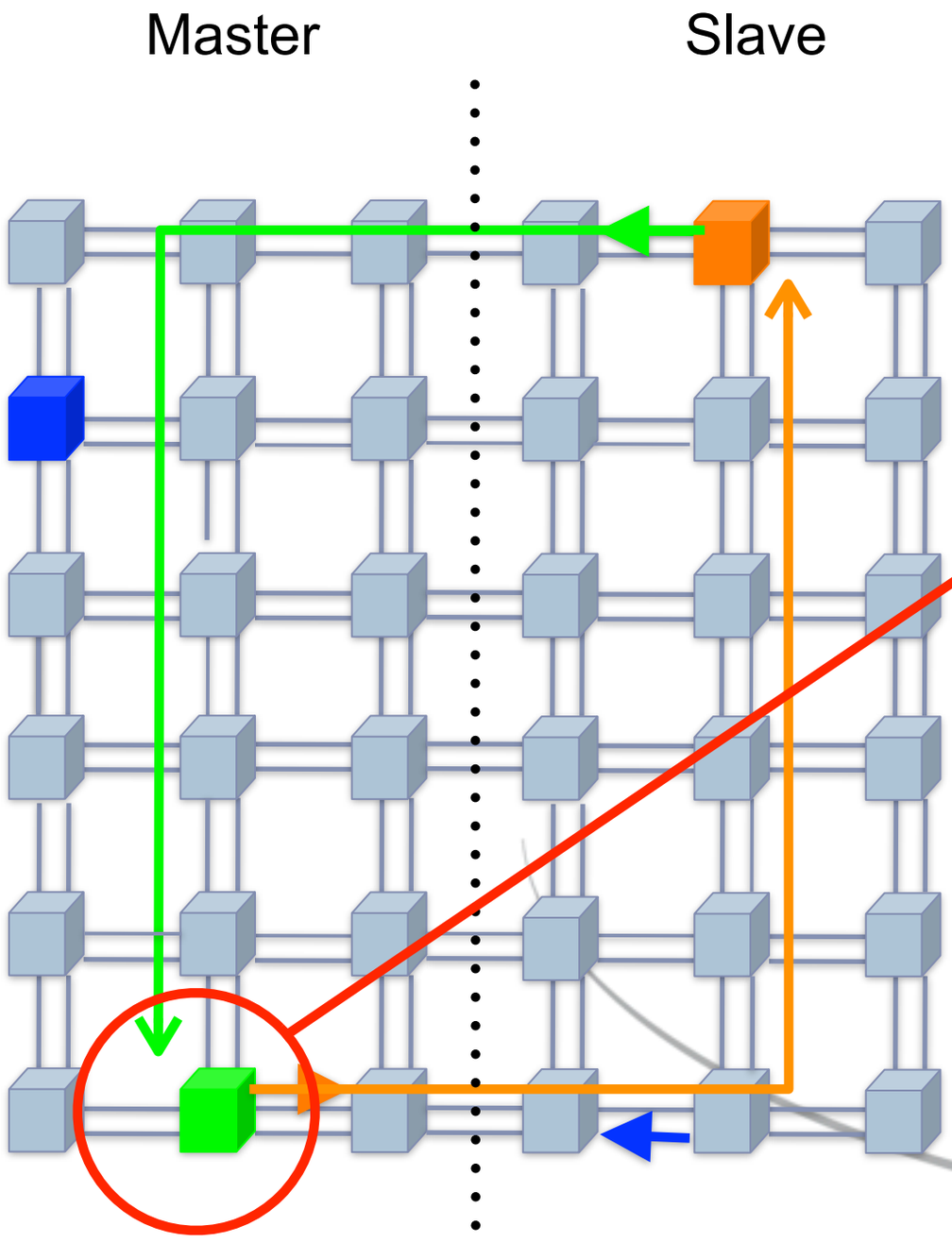
All masters on the right and all slaves on the left

- Is the system deadlock-free ?



All masters on the right and all slaves on the left

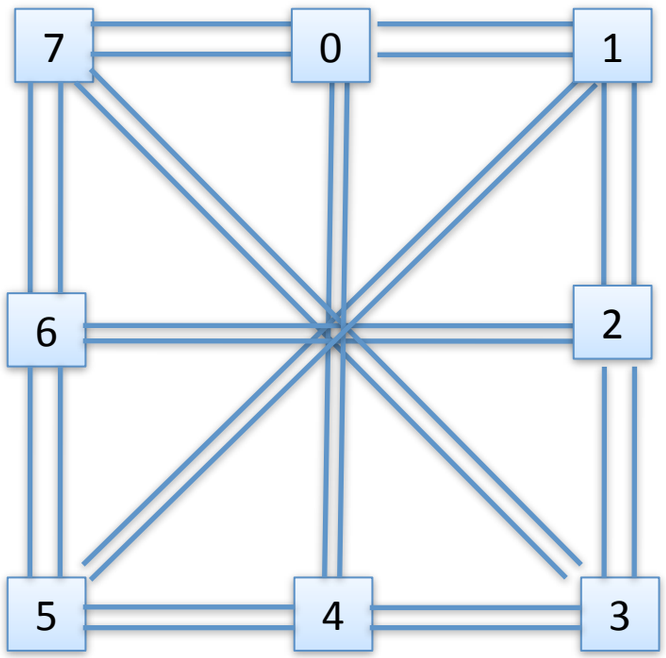
- Is the system deadlock-free ?
- Yes ! A deadlock would require a response to wait for a request



No dependencies between responses and requests

From deadlock-free components a deadlock emerges !

Networks-on-Chips: Example 2, Spidergon



RelAd = (dest - current) mod 4 * N

if RelAd = 0 **then**
    stop

elseif 0 < RelAd <= N **then**
    go clockwise

elseif 3*N <= RelAd <= 4*N **then**
    go counter clockwise

else
    go across

endif

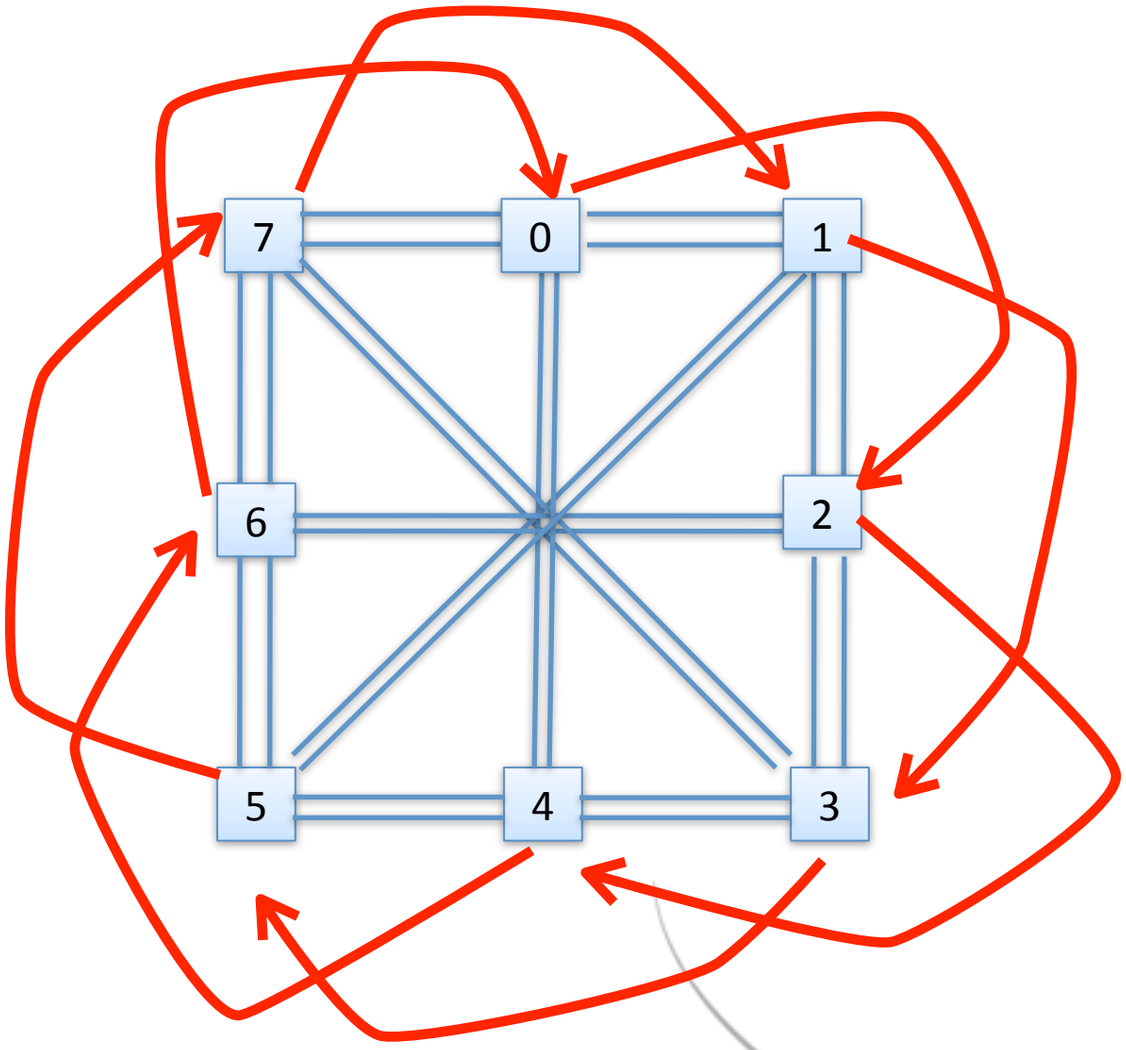
Route from 0 to 7 ? 1 to 6 ?

- Design by STMicroelectronics
- Simple shortest path routing algorithm
- Regular for an even number of nodes
- Packet, circuit, or wormhole switching

www.ou.nl

The interconnect has deadlocks !

- For instance, we can have a cycle of packets

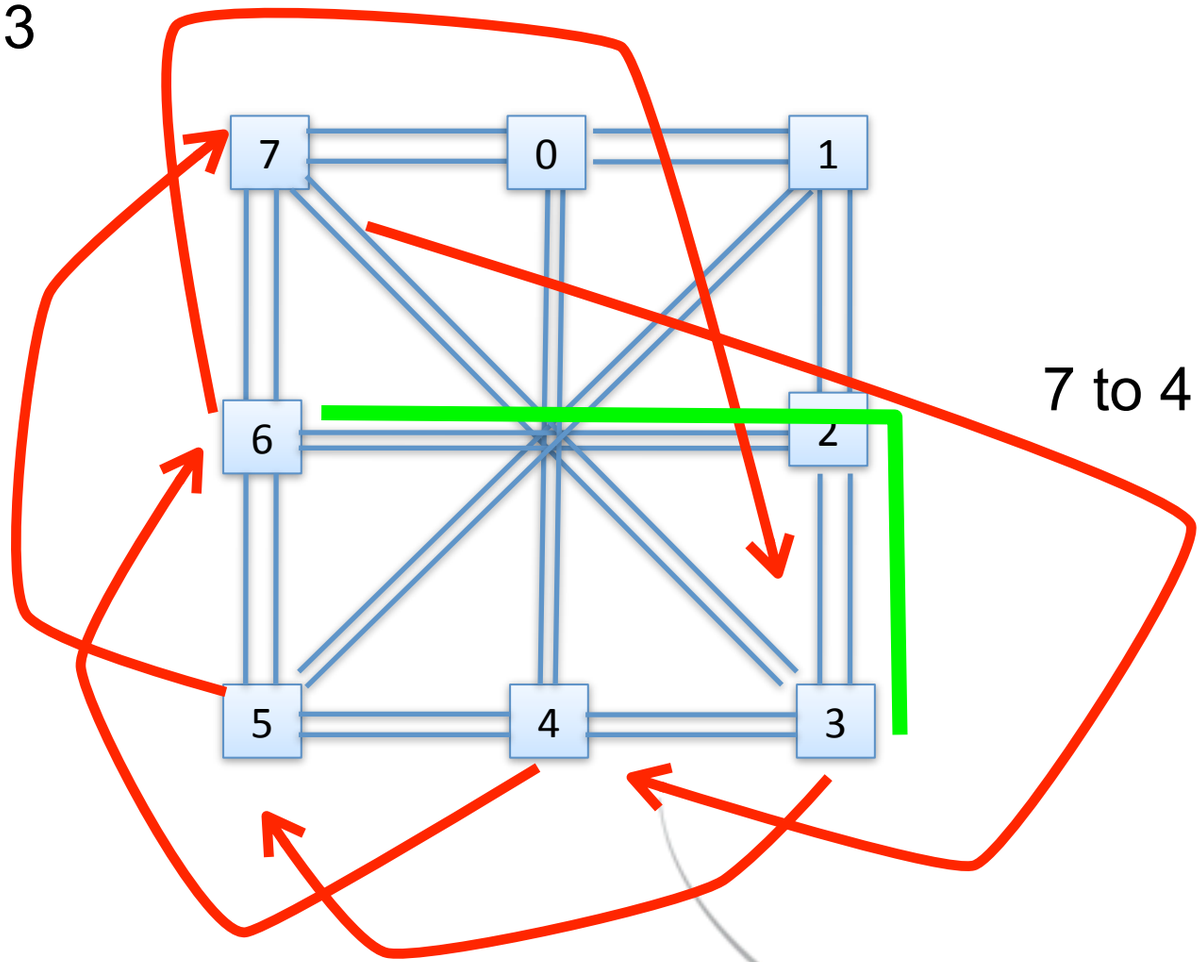


The only possible cycle is the ring

- Because routing is across first

6 to 3 route through 2 not 7

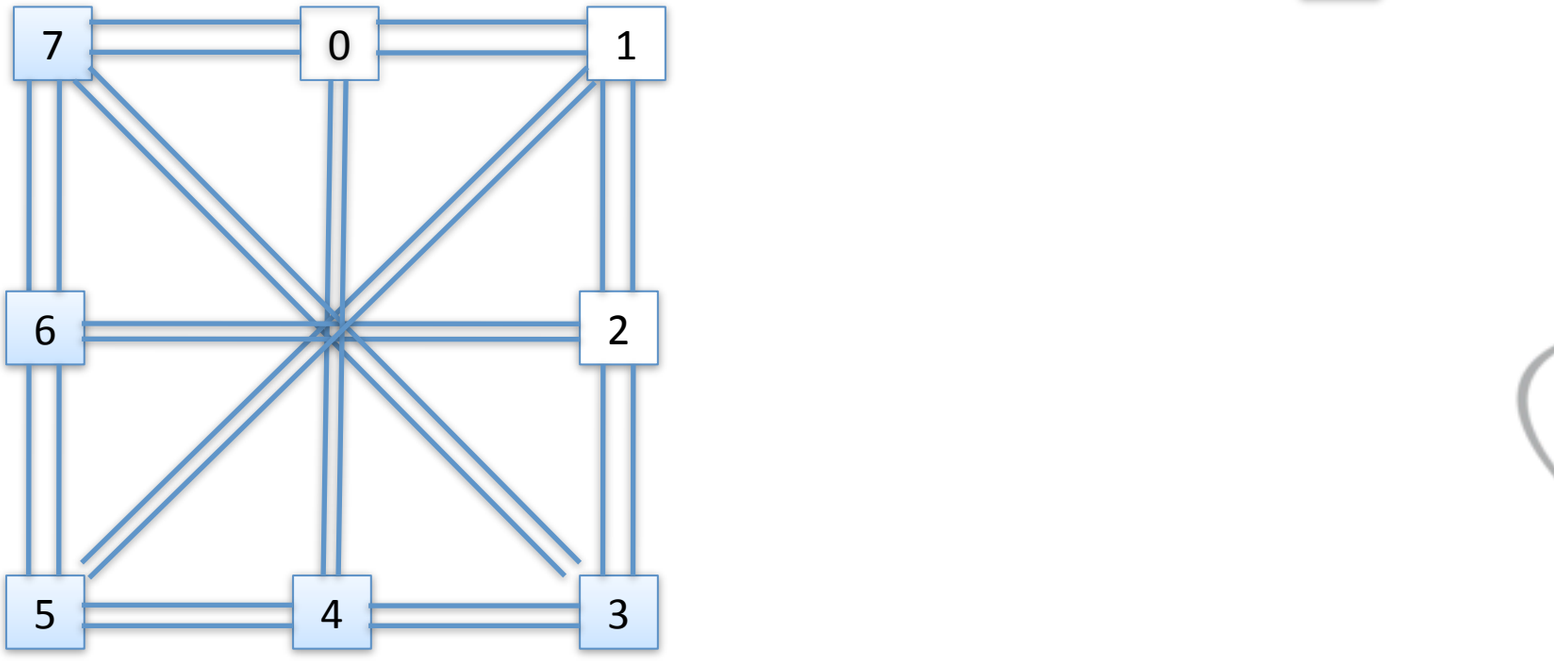
6 to 3

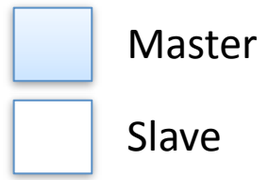


7 to 4

Nodes in first quarter are slaves only

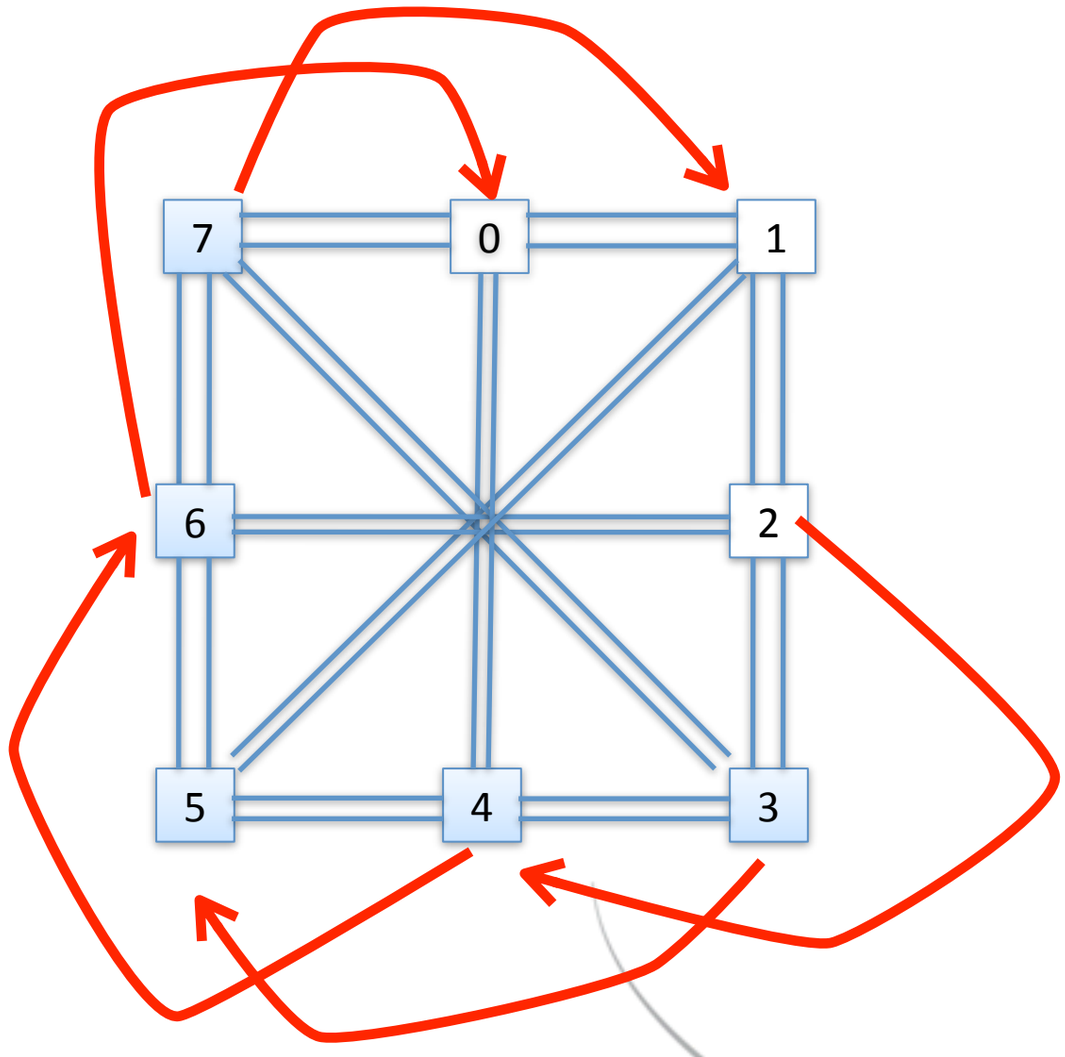
- Is the system deadlock-free ?

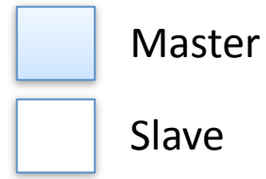




Nodes in first quarter are slaves only

- Is the system deadlock-free ?
- Yes ! A cycle would require a response going from 0 to 2.

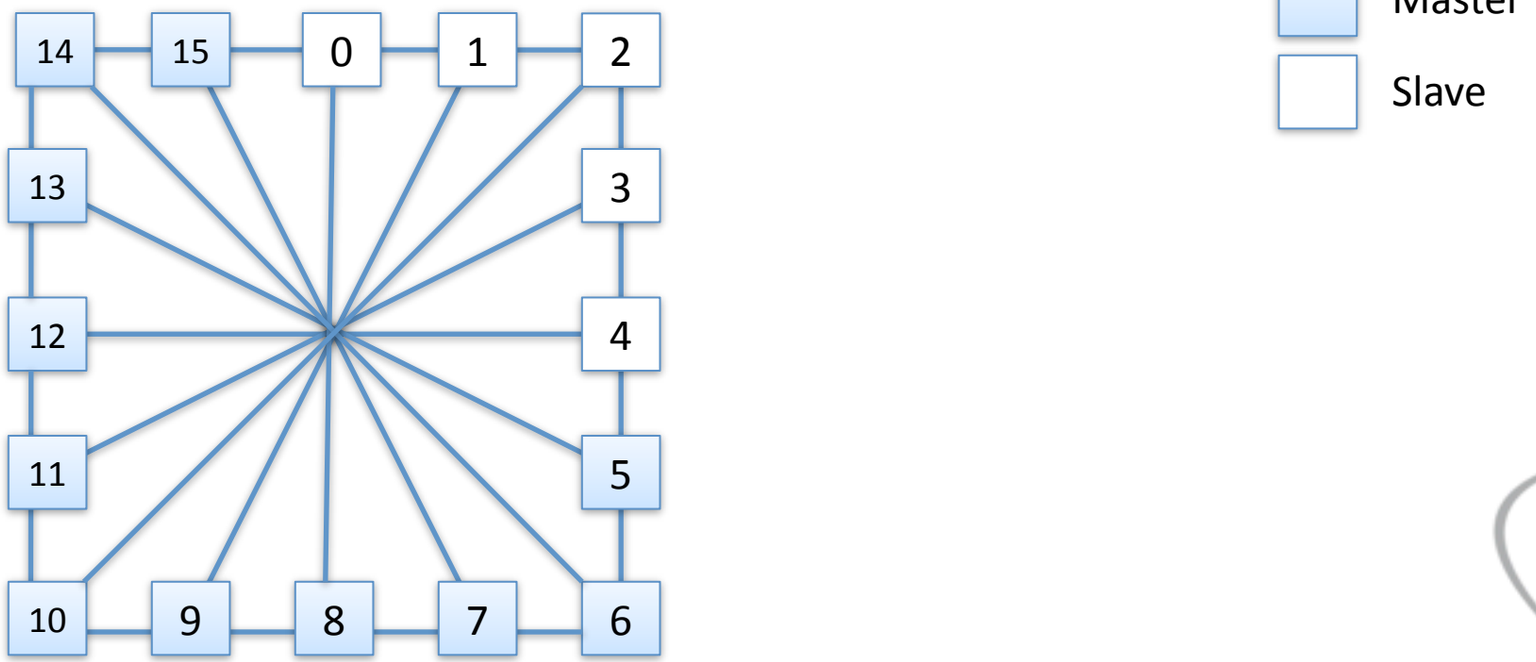




From components with deadlocks a deadlock-free system emerges !

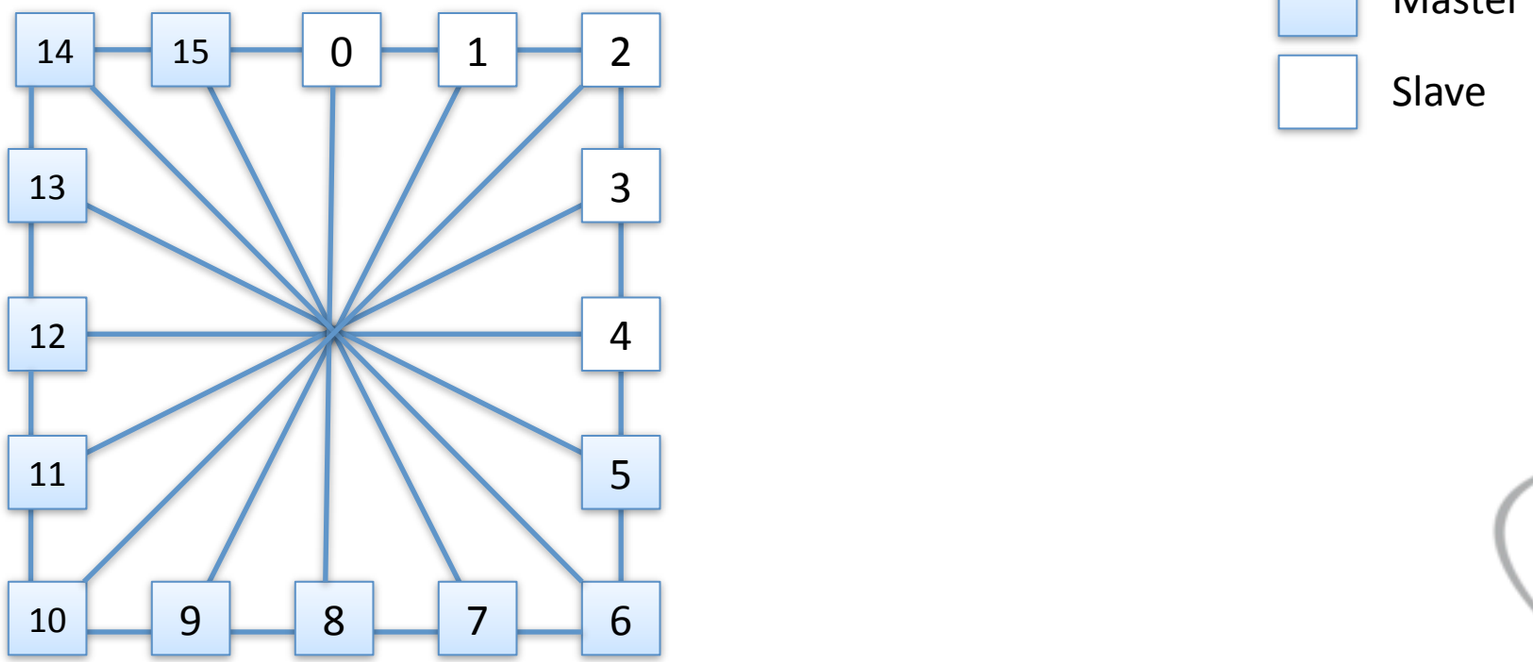
Nodes in first quarter are slaves only

- Is the system deadlock-free ?



Nodes in first quarter are slaves only

- Is the system deadlock-free ?
- No !



Confusing ...

- We need tools to (quickly) check for deadlocks
  - in large systems
  - with message dependencies

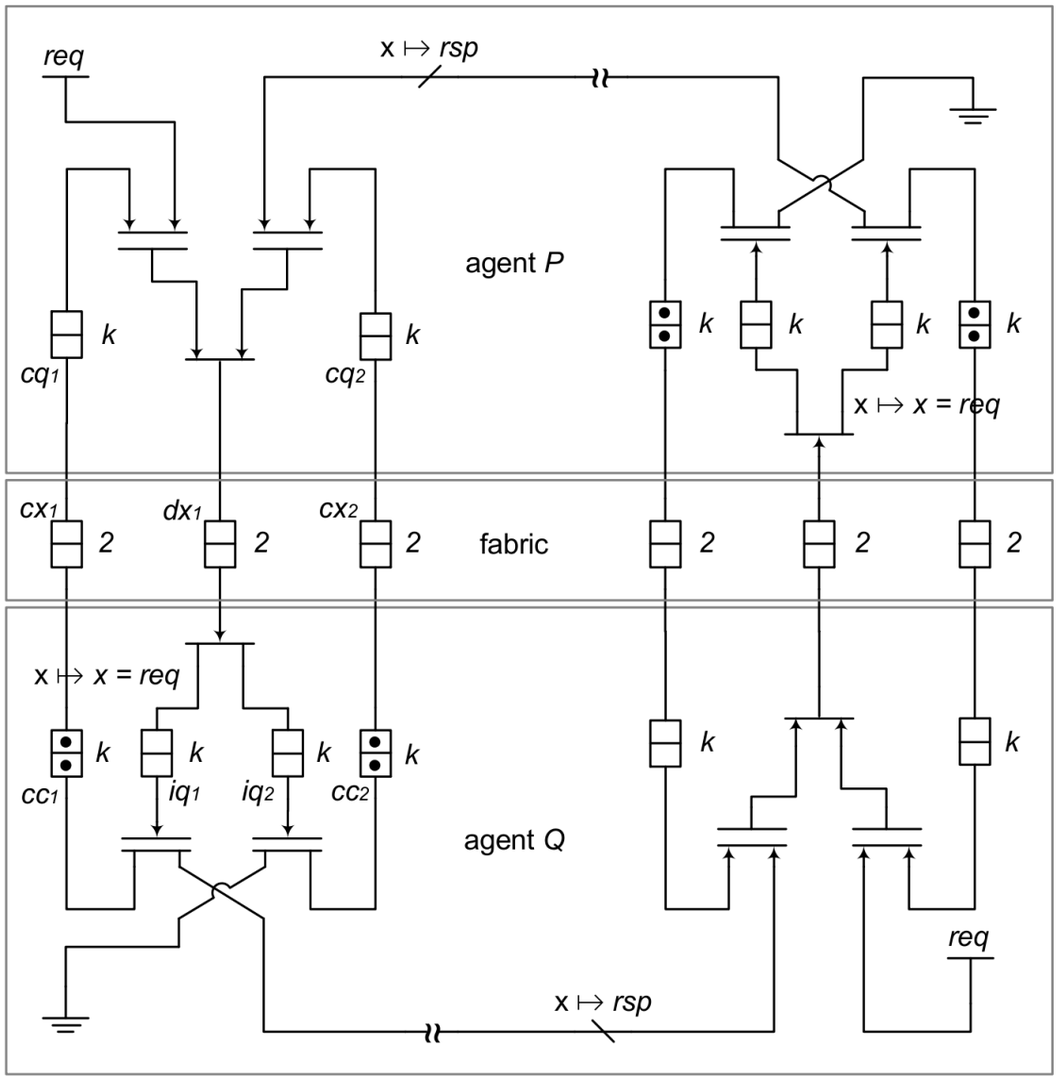
Outline

- Intel's micro-architectural description language
  - xMAS language
  - Capturing high-level structure and message dependencies

- Deadlock verification for xMAS
  - Definition of deadlocks
  - Labelled dependency graph
  - Feasible logically closed subgraph

- Conclusion and future work

Intel's abstraction for communication fabrics

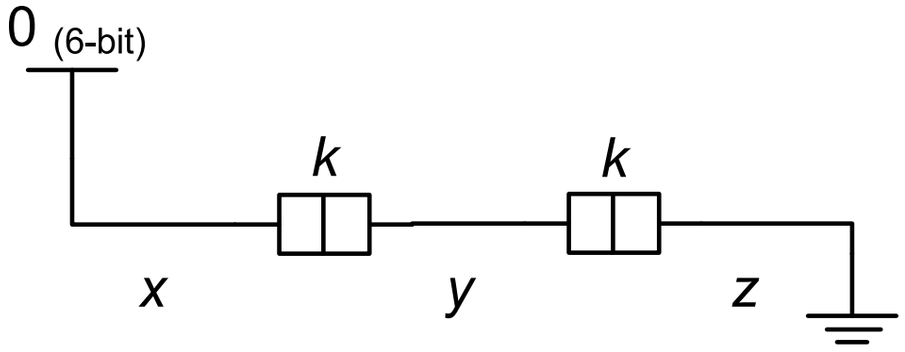


- High-level of abstraction
- Exploit high-level structure

Automatic proofs using invariant generation and hardware model-checking

www.ou.nl

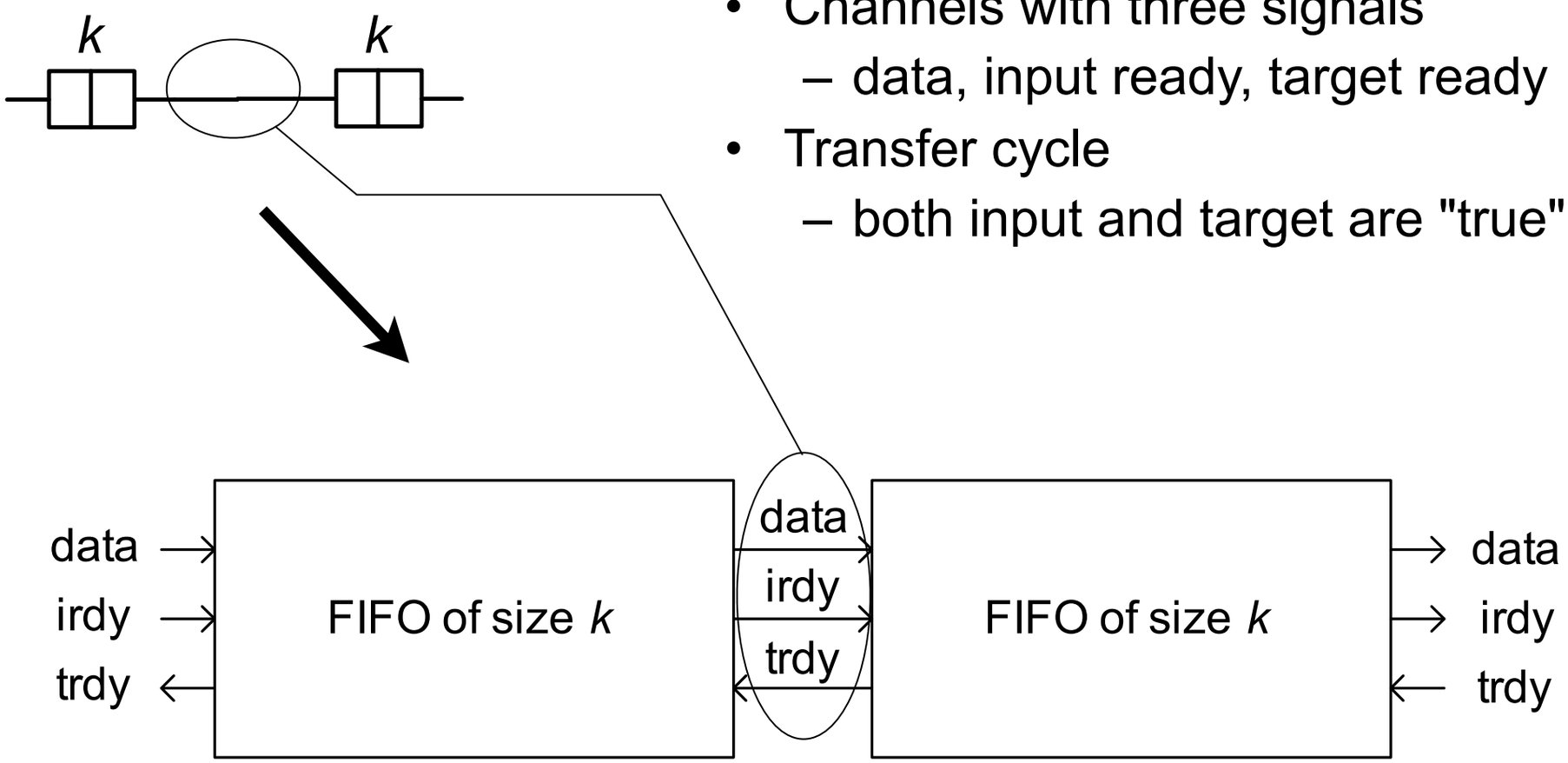
xMAS - Executable MicroArchitectural Specifications



- Fair sinks and sometimes sources
- Diagram is formal model
- Friendly to microarchitects

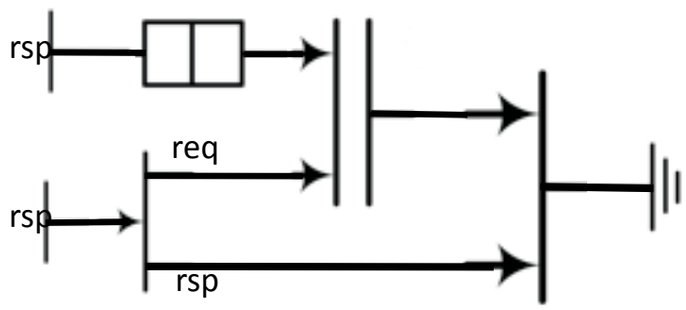
www.ou.nl

Composing modules via channels



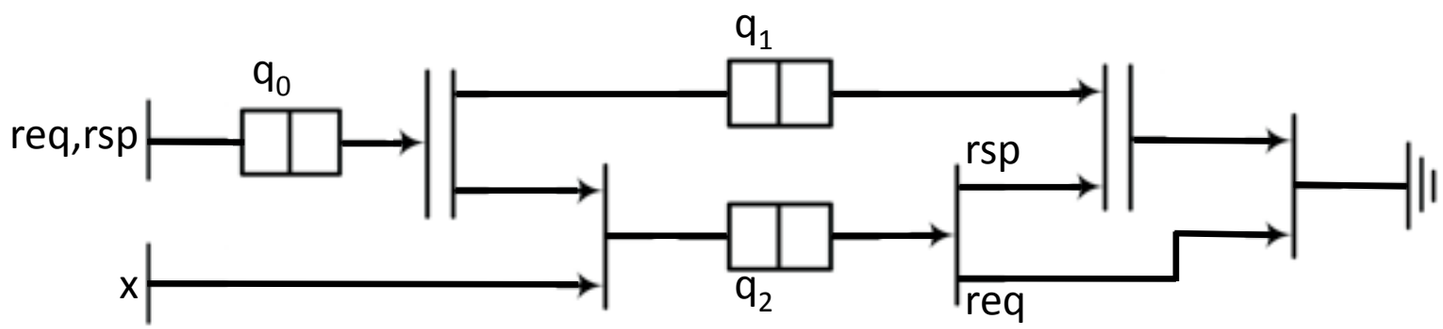
- Channels with three signals
  - data, input ready, target ready
- Transfer cycle
  - both input and target are "true"

A simple example



- Two sources
  - both inject responses

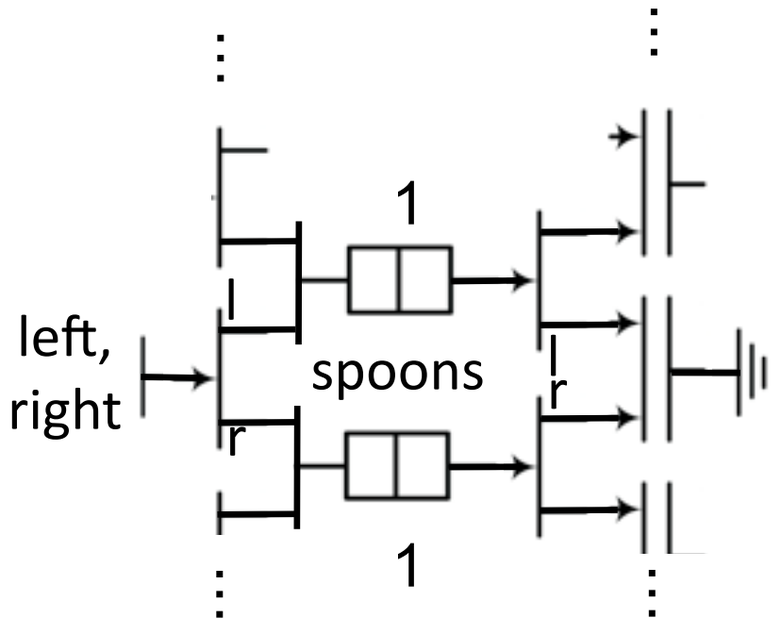
Another simple example



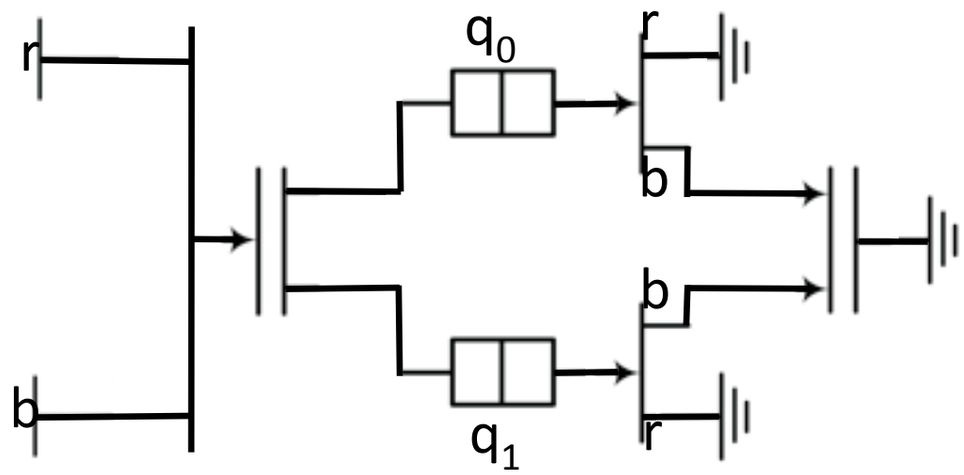
- Two message types
  - requests and responses

An academic example - Dining Philosophers

- Philosophers model in xMAS
  - Hands as 2 message types
  - Spoons as queues of size 1
  - "Eat" as join between hands

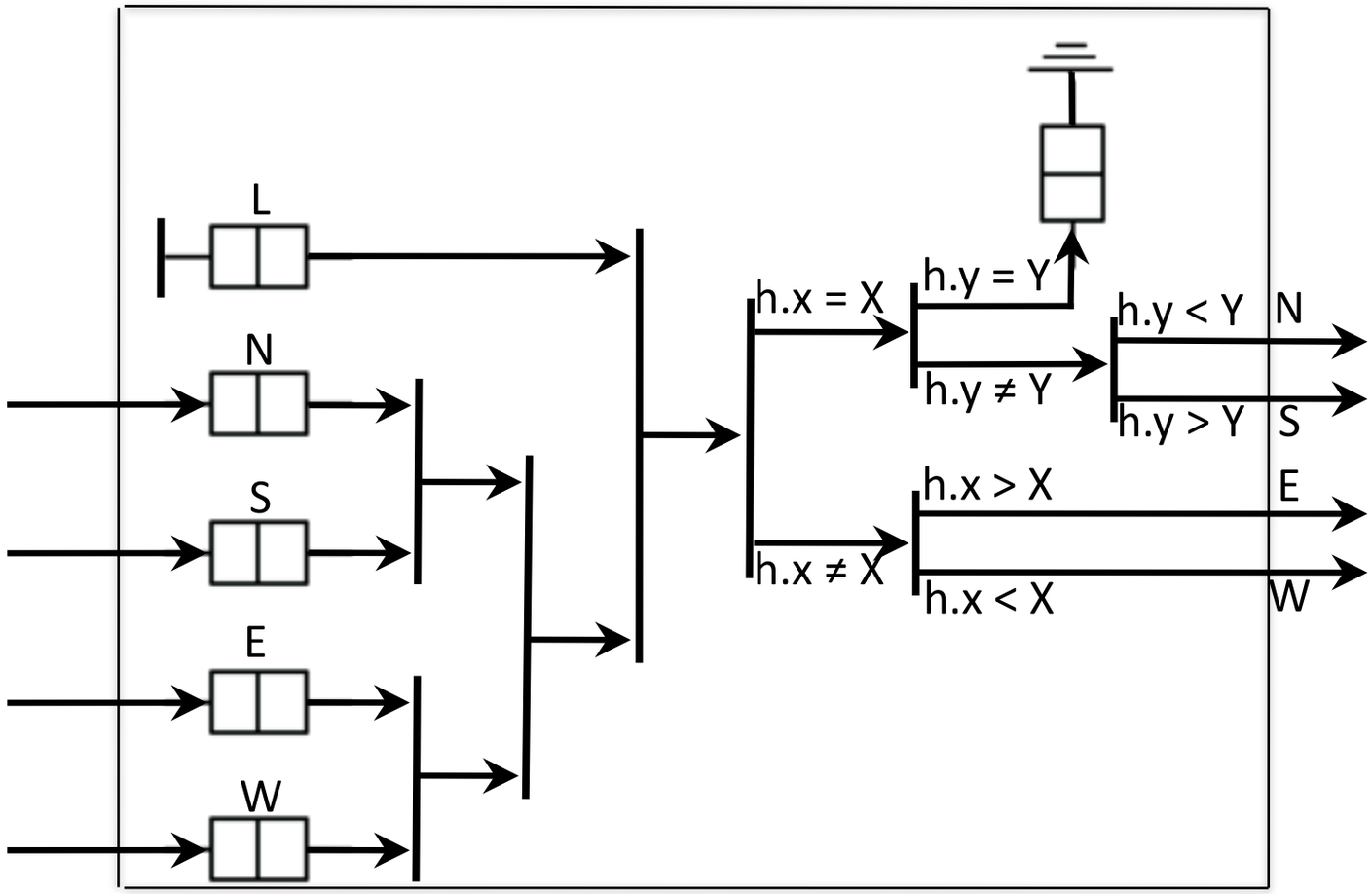


An example extracted from Intel's designs

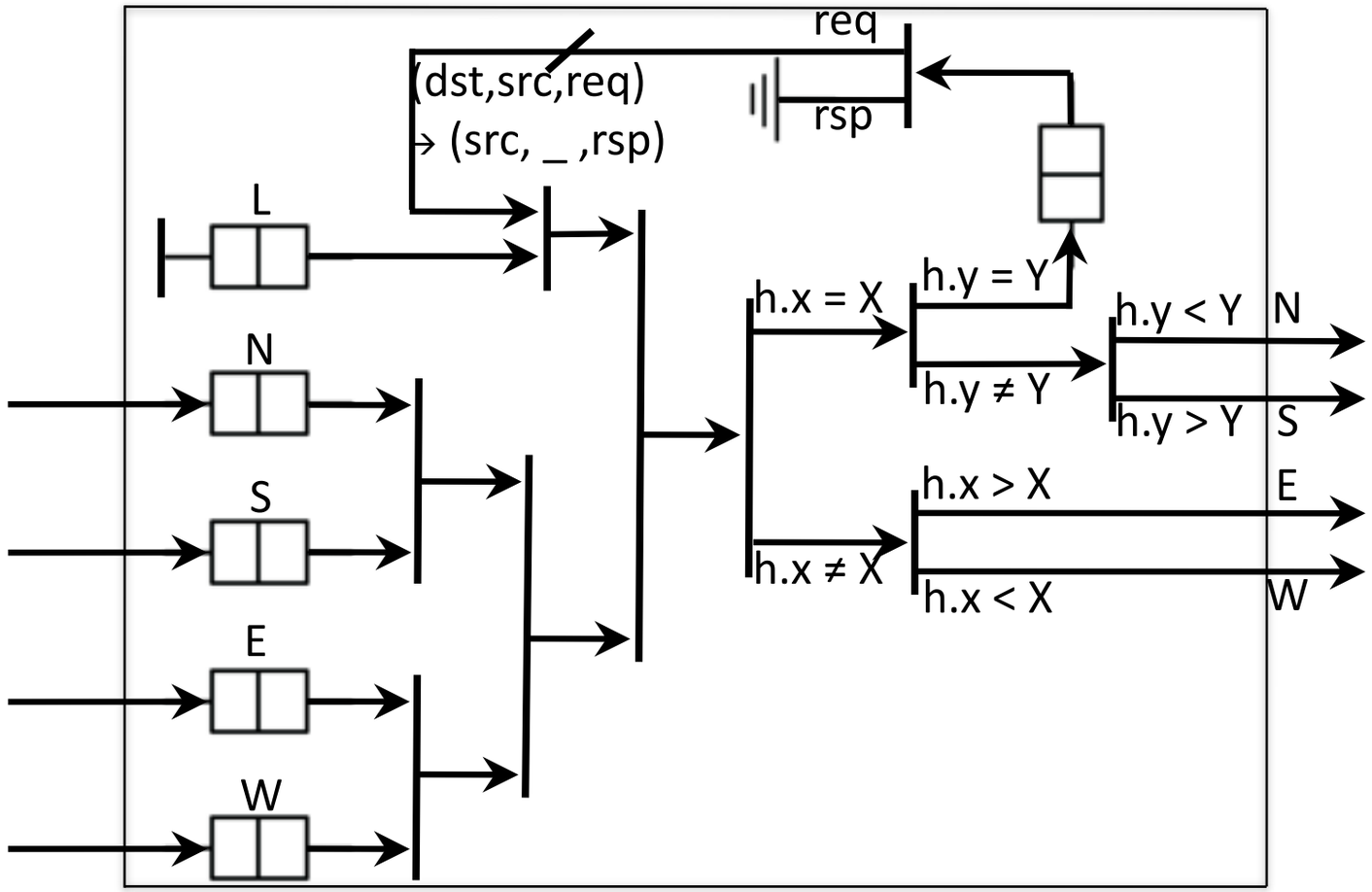


- Two message types
  - blue and red
- Sorting queues
- Reds and blues synchronized before sink

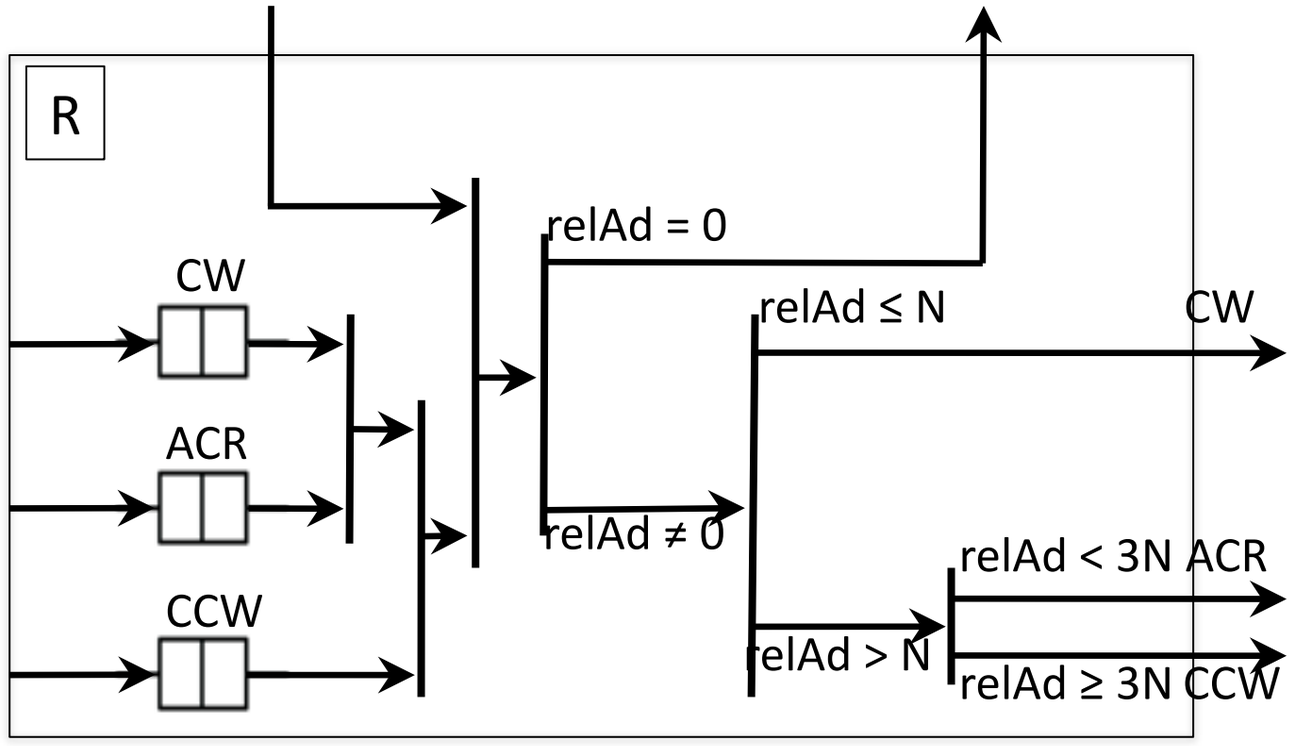
Processing node for XY routing in a 2D-mesh



Processing node with requests and responses



Processing node for Spidergon



Outline

- Intel's micro-architectural description language
  - xMAS language
  - Capturing high-level structure and message dependencies

- Deadlock verification for xMAS
  - Definition of deadlocks
  - Labelled dependency graph
  - Feasible logically closed subgraph

- Conclusion and future work

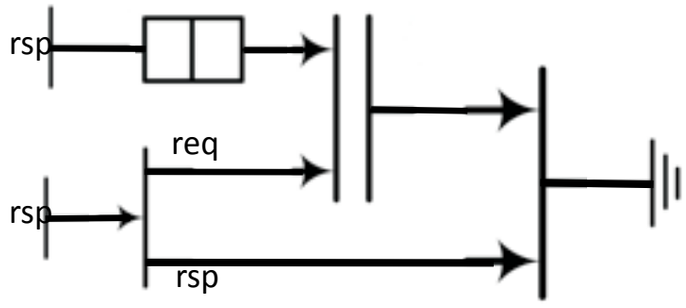
Formal definition of "deadlock" in xMAS

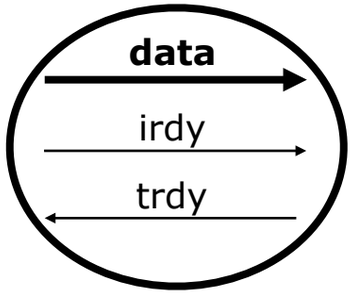
- Intuition is a "dead" channel

- Formal definition based on Linear Temporal Logic
  - Predicate logic
  - Temporal operators "eventually" (F) and "globally" (G)

- Channel u is dead iff
  - **F (u.irdy => G ~u.trdy)**
  - Eventually the input is ready and the target is globally (forever) not ready
  - A packet arrives at a channel but will never be able to cross it

A simple example



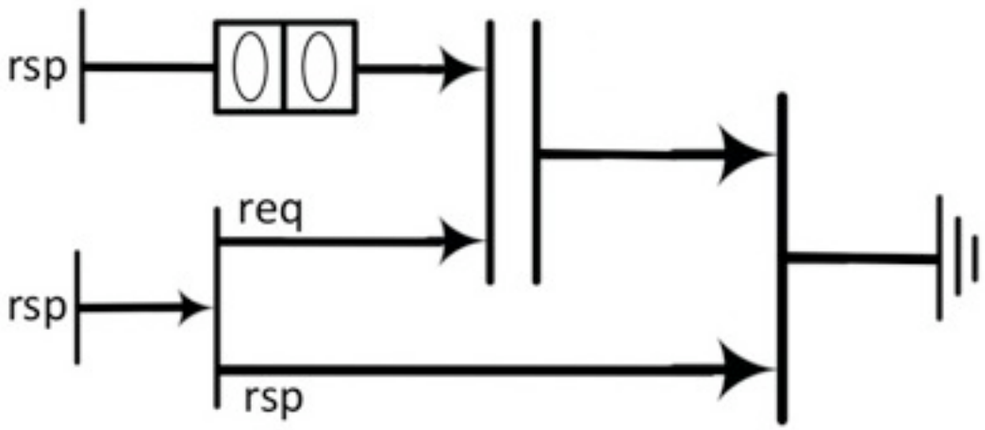


- Two sources
  - one for requests
  - one for responses

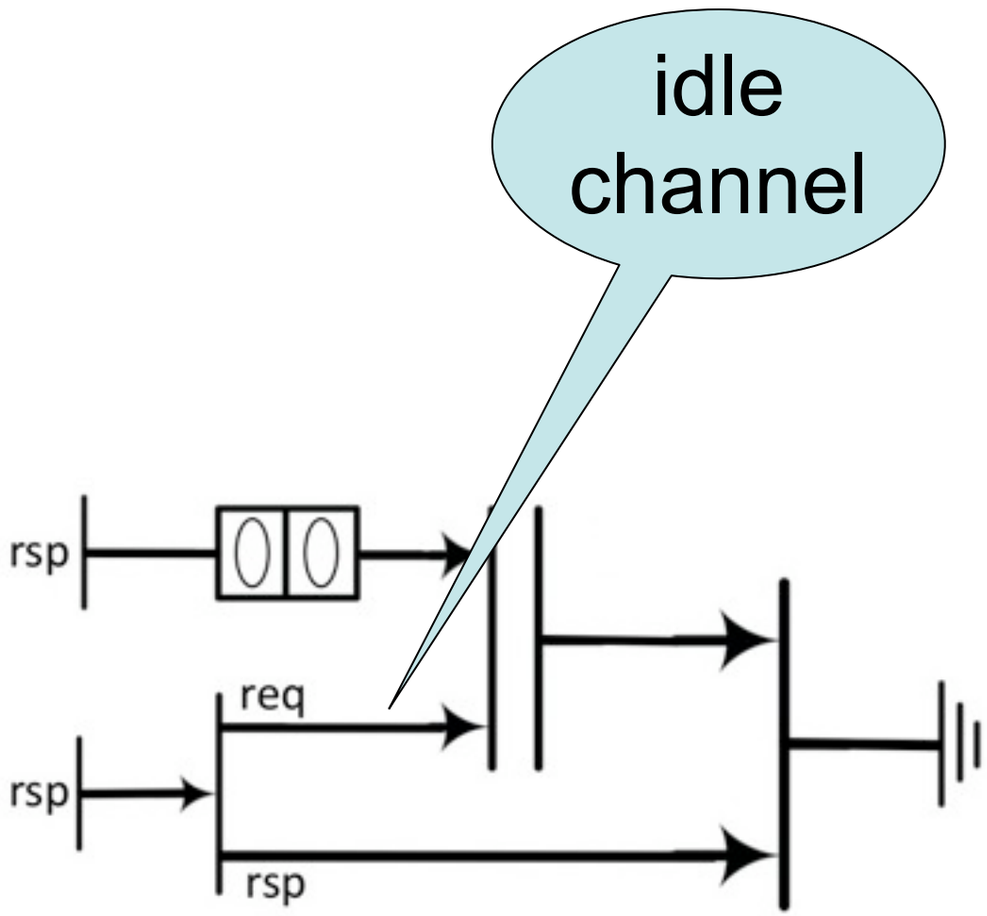
- Is it deadlock-free ?

A simple example

- Two sources for responses
- There is a deadlock
  - no cycle
  - no message dependencies

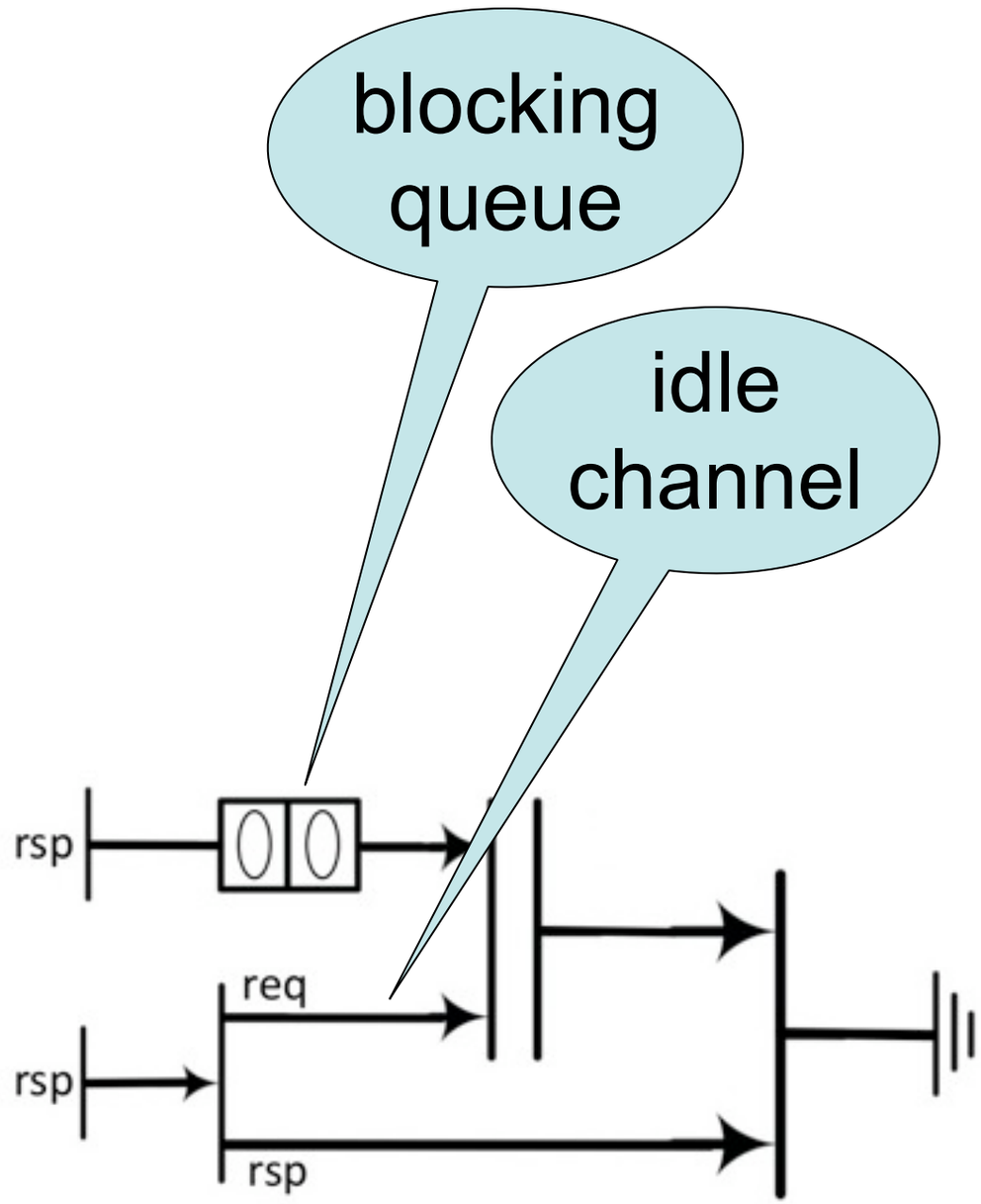


A simple example



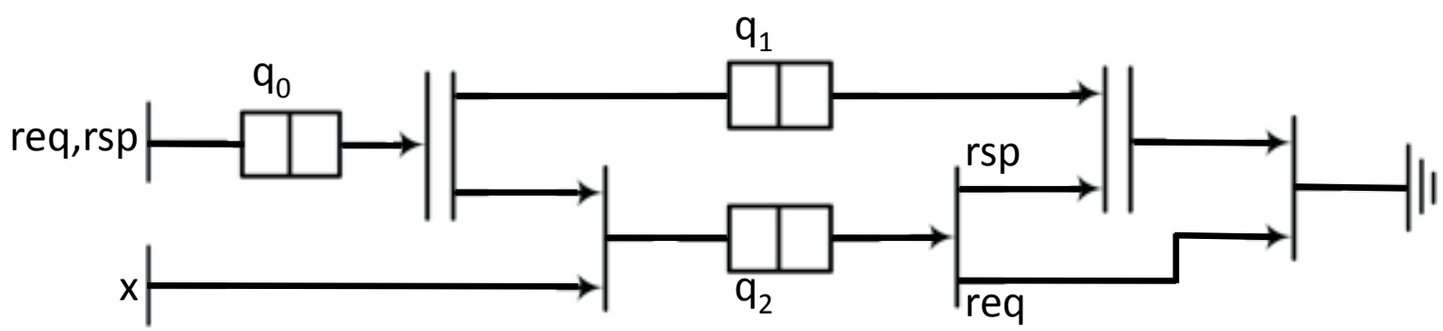
- Two sources for responses
- There is a deadlock
  - no cycle
  - no message dependencies

A simple example



- Two sources for responses
- There is a deadlock
  - no cycle
  - no message dependencies

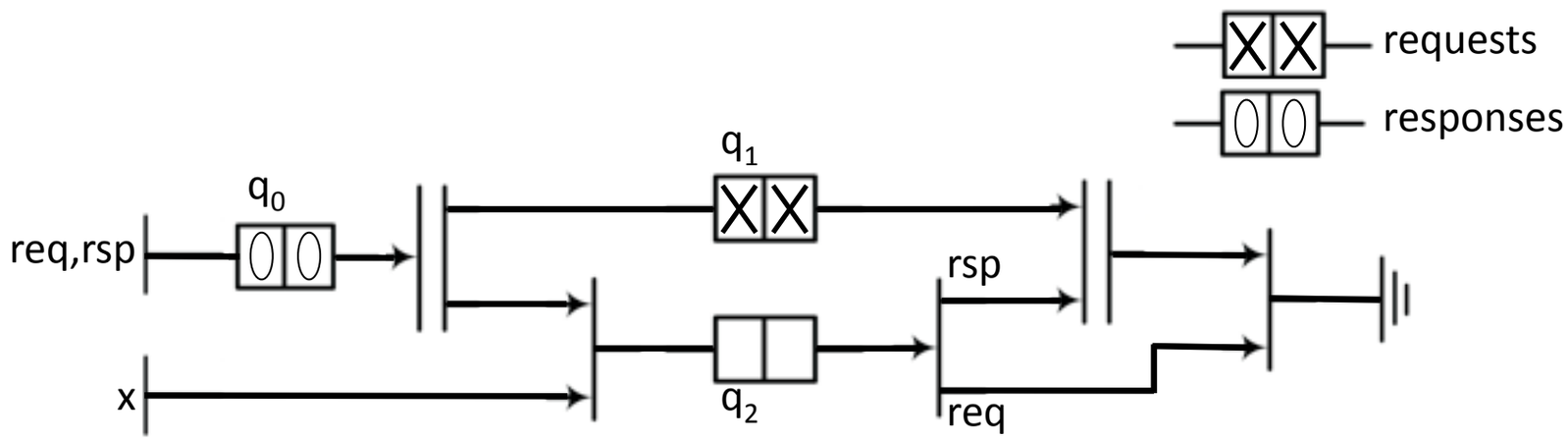
Another simple example



- Two message types
  - requests and responses

- Types at source x without creating a deadlock ?

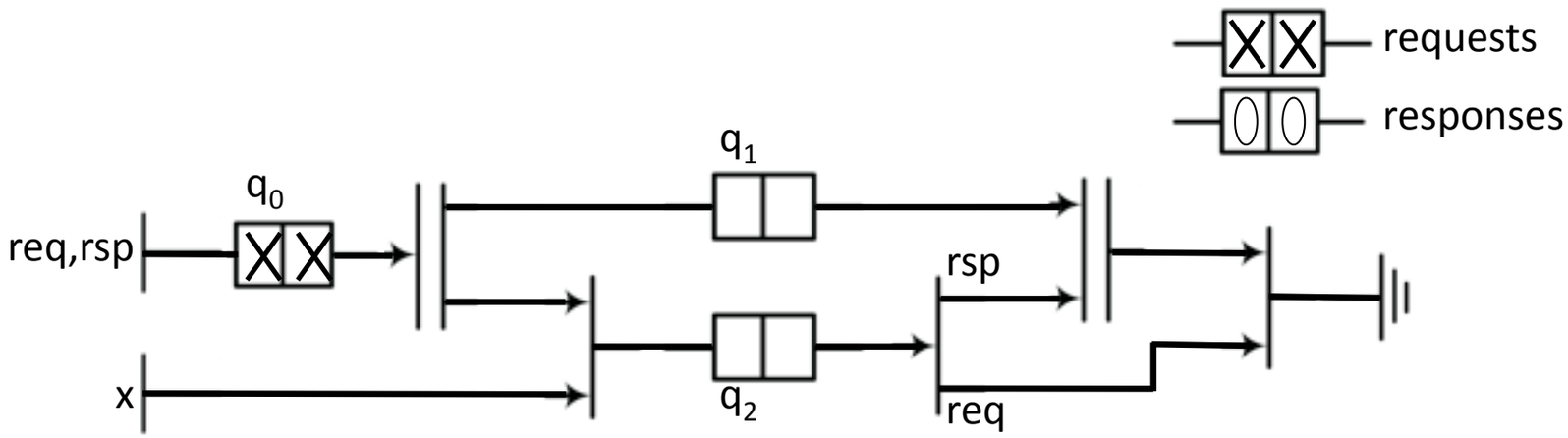
Another simple example - deadlock configuration



- Two message types
  - requests and responses

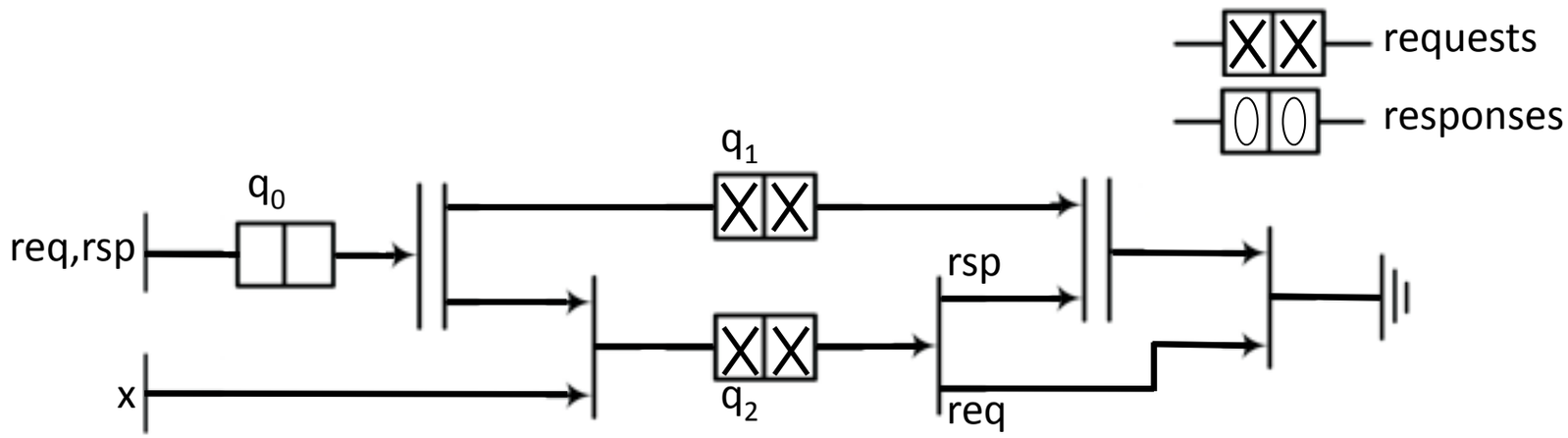
- Types at source x without creating a deadlock ?
  - If x = rsp no deadlock
  - If x = req then requests get blocked in q1

Another simple example - deadlock configuration (1)



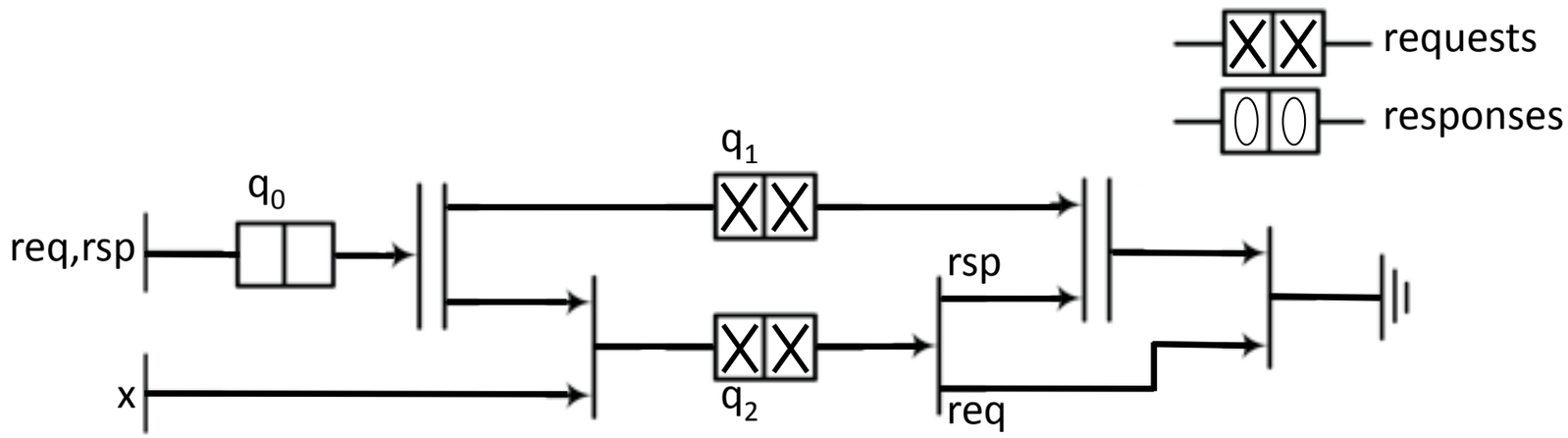
- Inject two requests in q0

Another simple example - deadlock configuration (2)



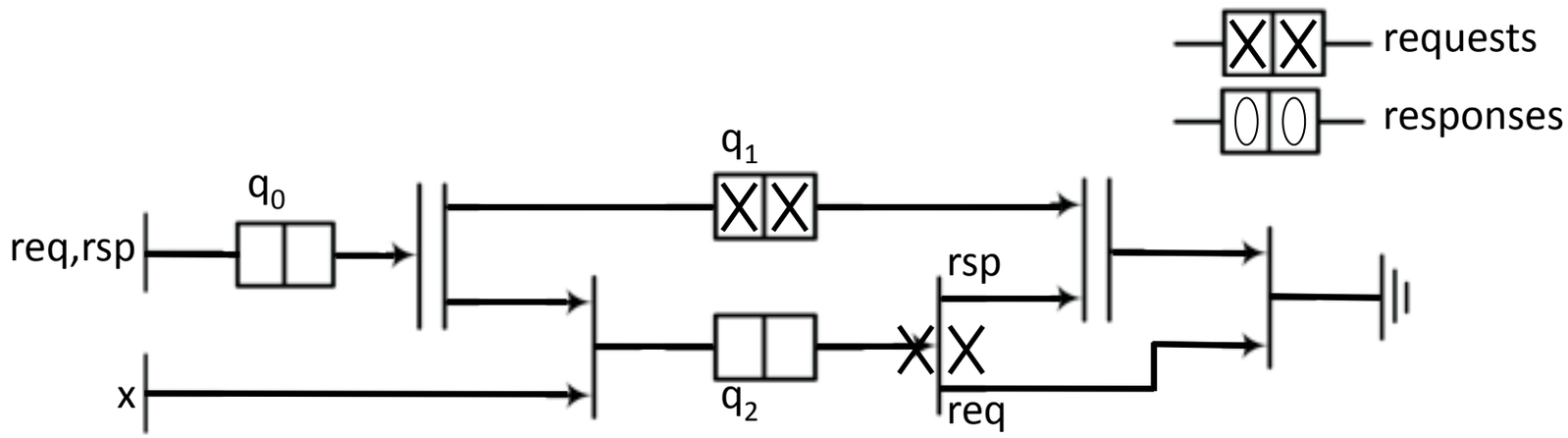
- Inject two requests in q0
- Fork creates two copies

Another simple example - deadlock configuration (3)



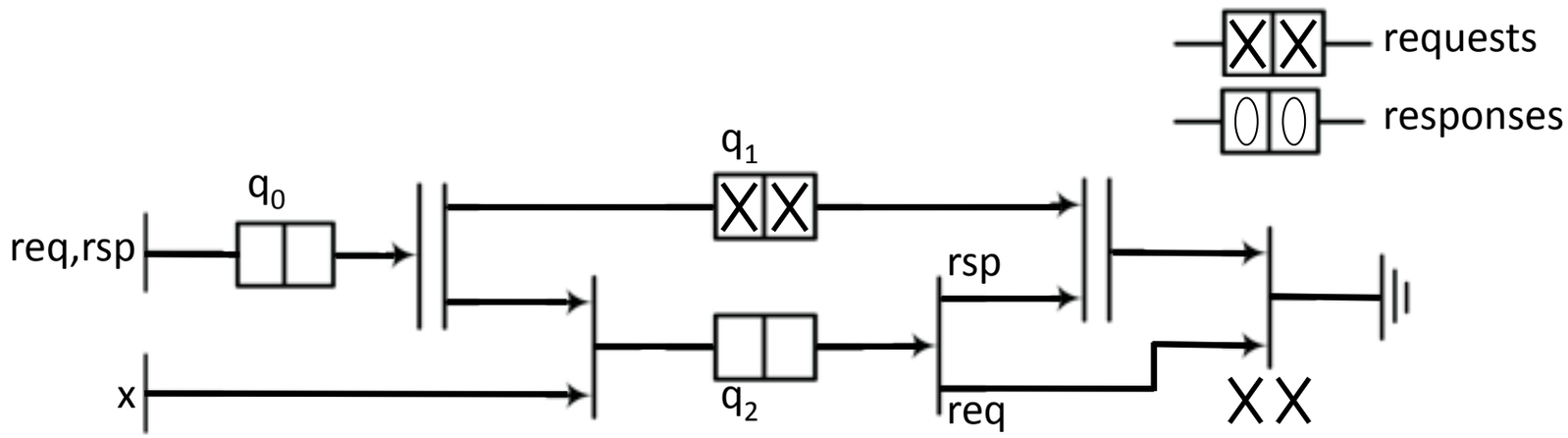
- Inject two requests in q0
- Fork creates two copies
- One pair is sunk

Another simple example - deadlock configuration (3)



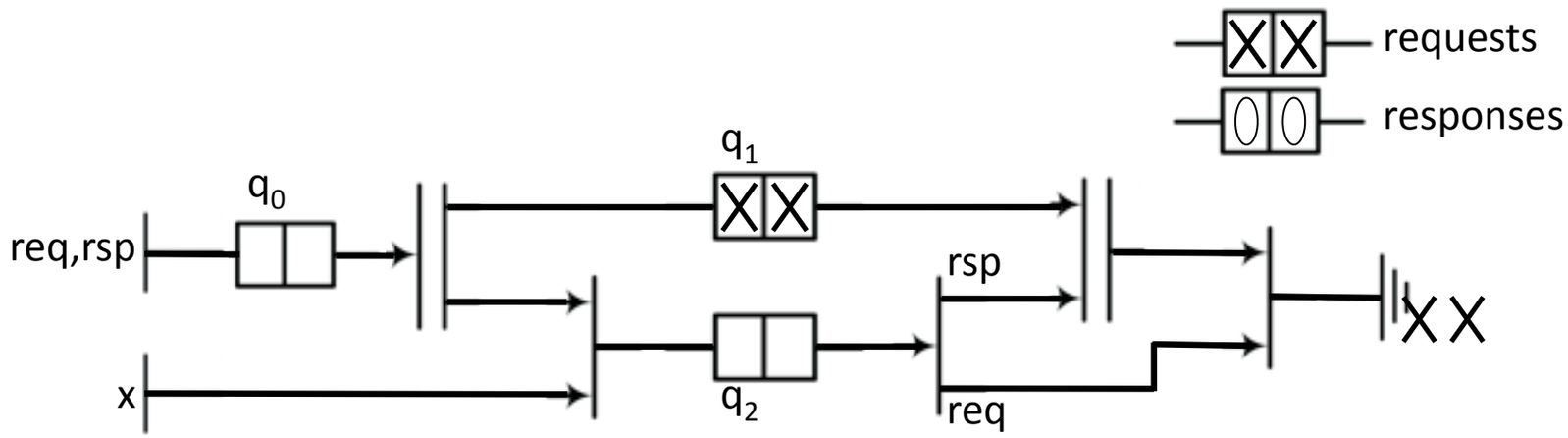
- Inject two requests in q0
- Fork creates two copies
- One pair is sunk

Another simple example - deadlock configuration (3)



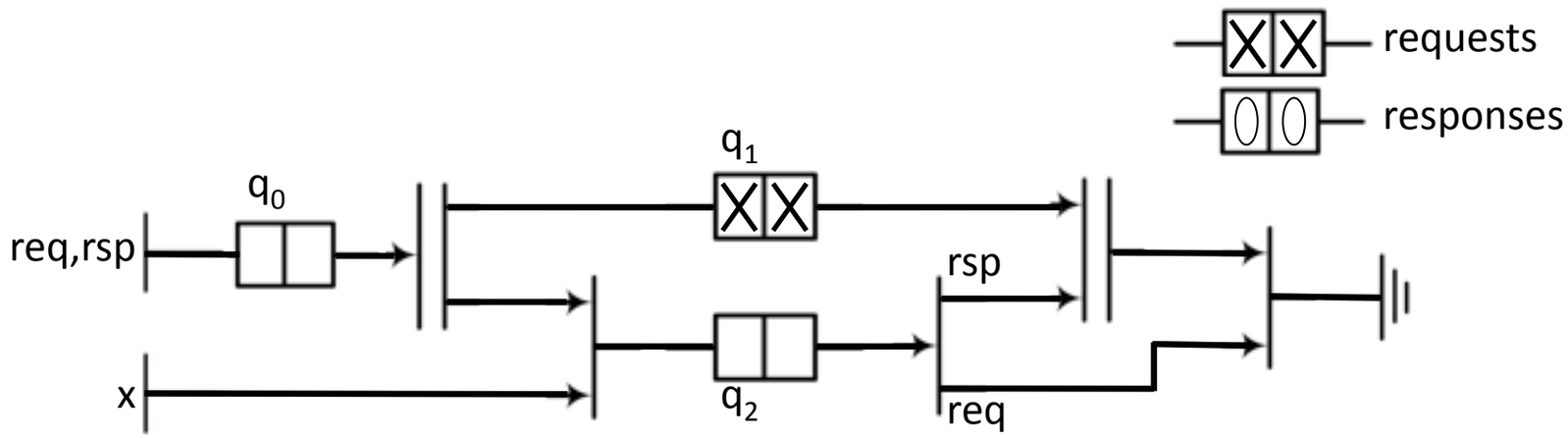
- Inject two requests in q0
- Fork creates two copies
- One pair is sunk

Another simple example - deadlock configuration (3)



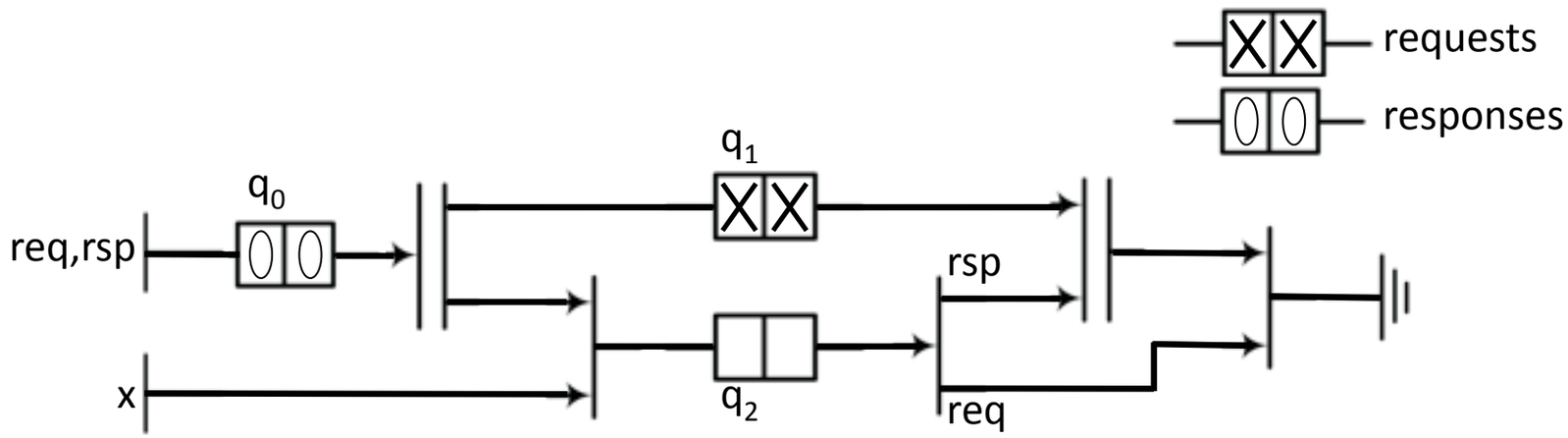
- Inject two requests in q0
- Fork creates two copies
- One pair is sunk

Another simple example - deadlock configuration (3)



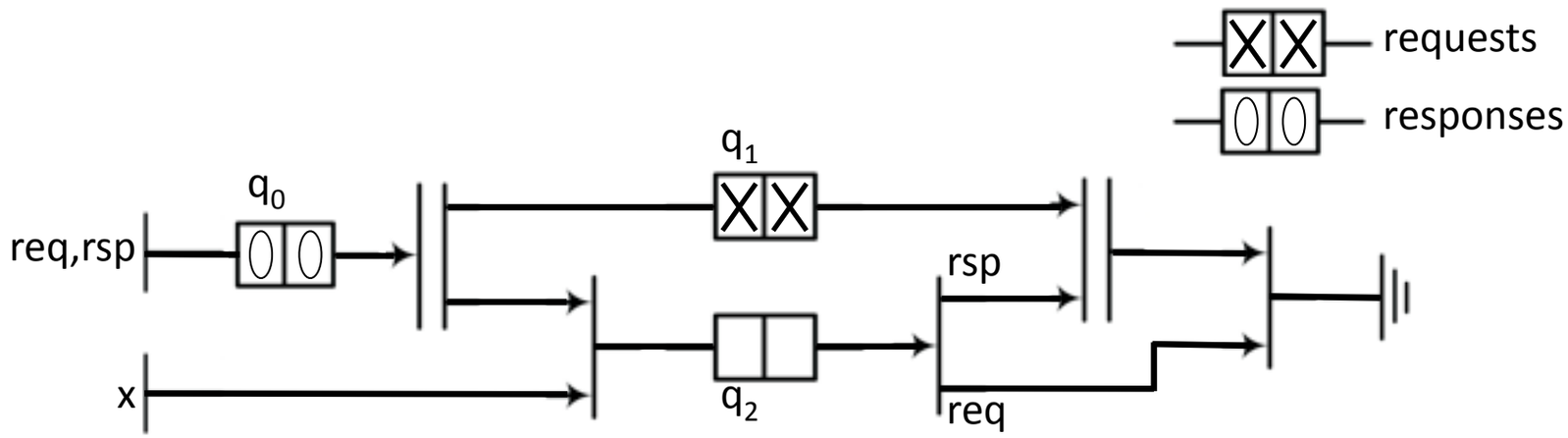
- Inject two requests in q0
- Fork creates two copies
- One pair is sunk

Another simple example - deadlock configuration (4)



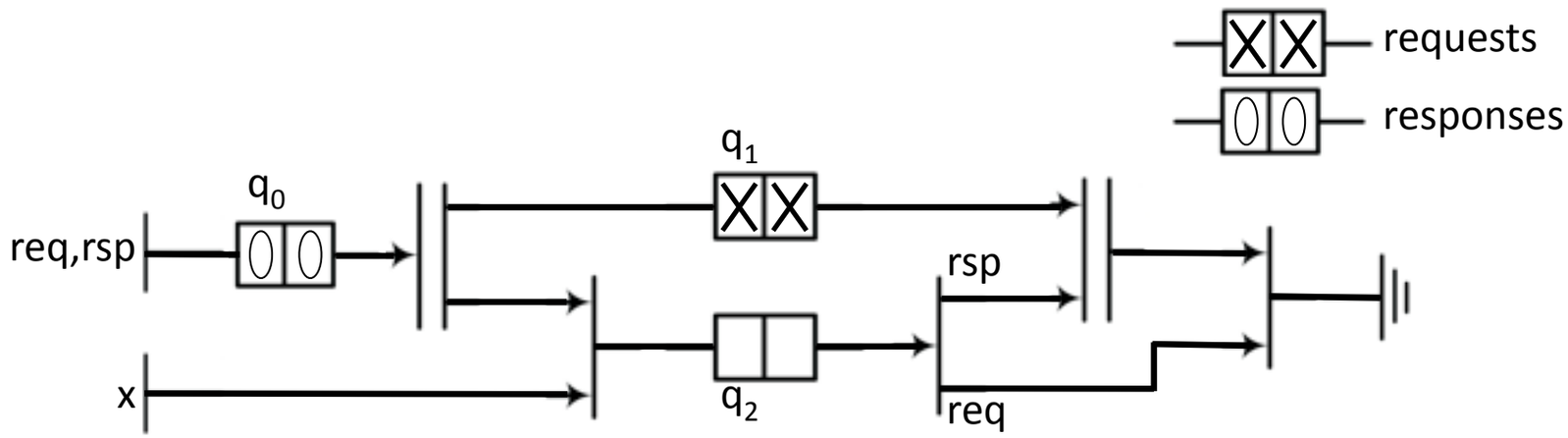
- Inject two requests in q0
- Fork creates two copies
- One pair is sunk
- Inject two responses in q0

Another simple example - deadlock configuration (5)



- Inject two requests in q0
- Fork creates two copies
- One pair is sunk
- Inject two responses in q0
- If x never injects responses, q1 is blocking

Another simple example - deadlock configuration (5)



- Inject two requests in q0
- Fork creates two copies
- One pair is sunk
- Inject two responses in q0
- If x never injects responses, q1 is blocking

We have a deadlock without a circular wait !

General approach for deadlock detection in xMAS networks

- Define deadlock equations for all components
  - Equations capture the reason why a component is idle or blocking

- Build a labelled waiting graph for each queue
  - Labels correspond to the equations
  - Graph captures the topology, i.e., the dependencies between the xMAS components

- Search for a feasible logically closed subgraph
  - Corresponds to a deadlock situation
  - Feasibility checked using Linear Programming

- This approach may output unreachable deadlocks
  - A first step generates invariants to rule out false deadlocks
  - Invariants are rather weak and simple - false deadlocks are in theory still possible

General approach for deadlock detection in xMAS networks

- Define deadlock equations for all components
  - Equations capture the reason why a component is idle or blocking

- Build a labelled waiting graph for each queue
  - Labels correspond to the equations
  - Graph captures the topology, i.e., the dependencies between the xMAS components

- Search for a feasible logically closed subgraph
  - Corresponds to a deadlock situation
  - Feasibility checked using Linear Programming

- This approach may output unreachable deadlocks
  - A first step generates invariants to rule out false deadlocks
  - Invariants are rather weak and simple - false deadlocks are in theory still possible

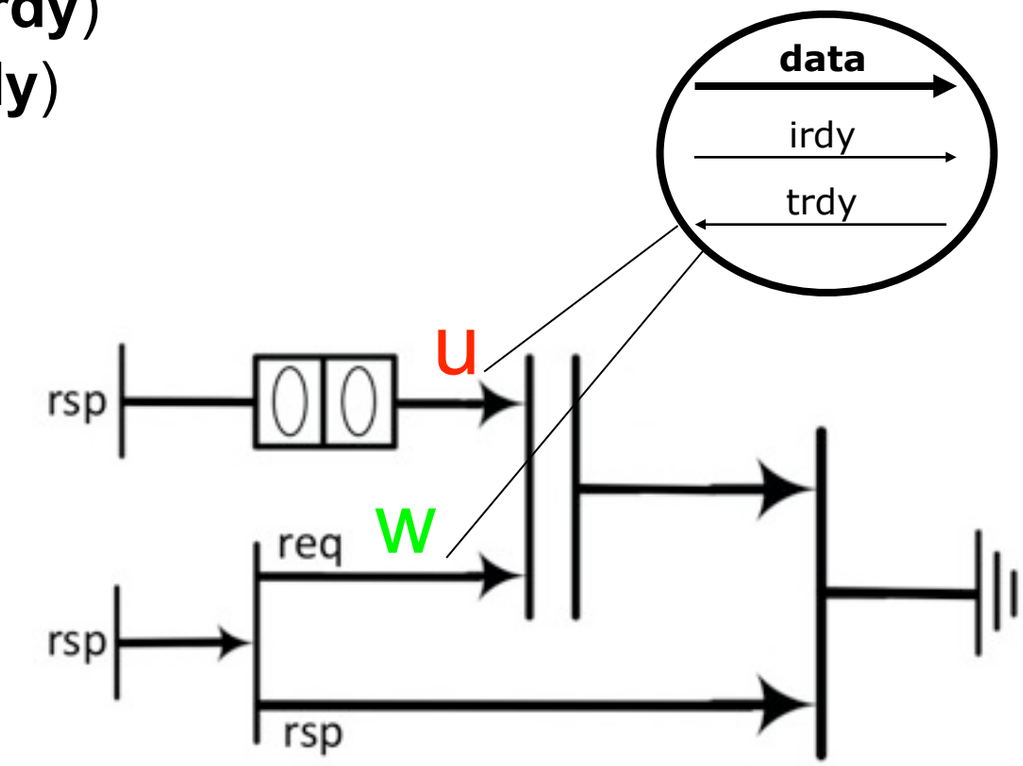
Idle and blocked channels - searching for deadlocks

- Definition of a deadlock
  - **F (u.irdy => G~u.trdy)**

- Two reasons for a deadlock
  - a blocked channel (**G~u.trdy**)
  - an "idle" channel (**G~u.irdy**)

v is blocked

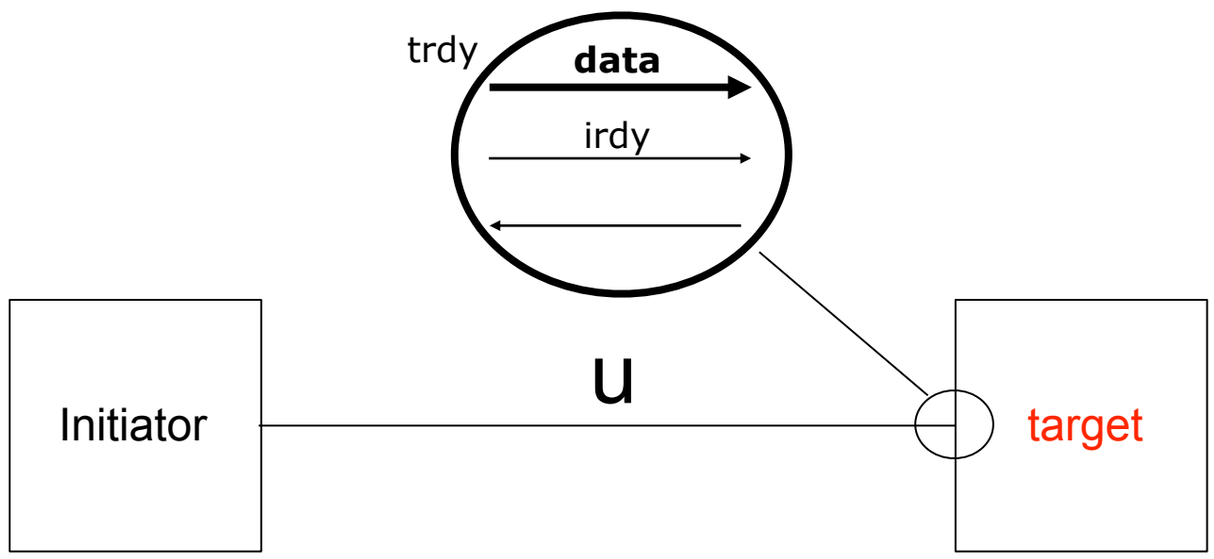
w is idle



www.ou.nl

Deadlock equations for a channel

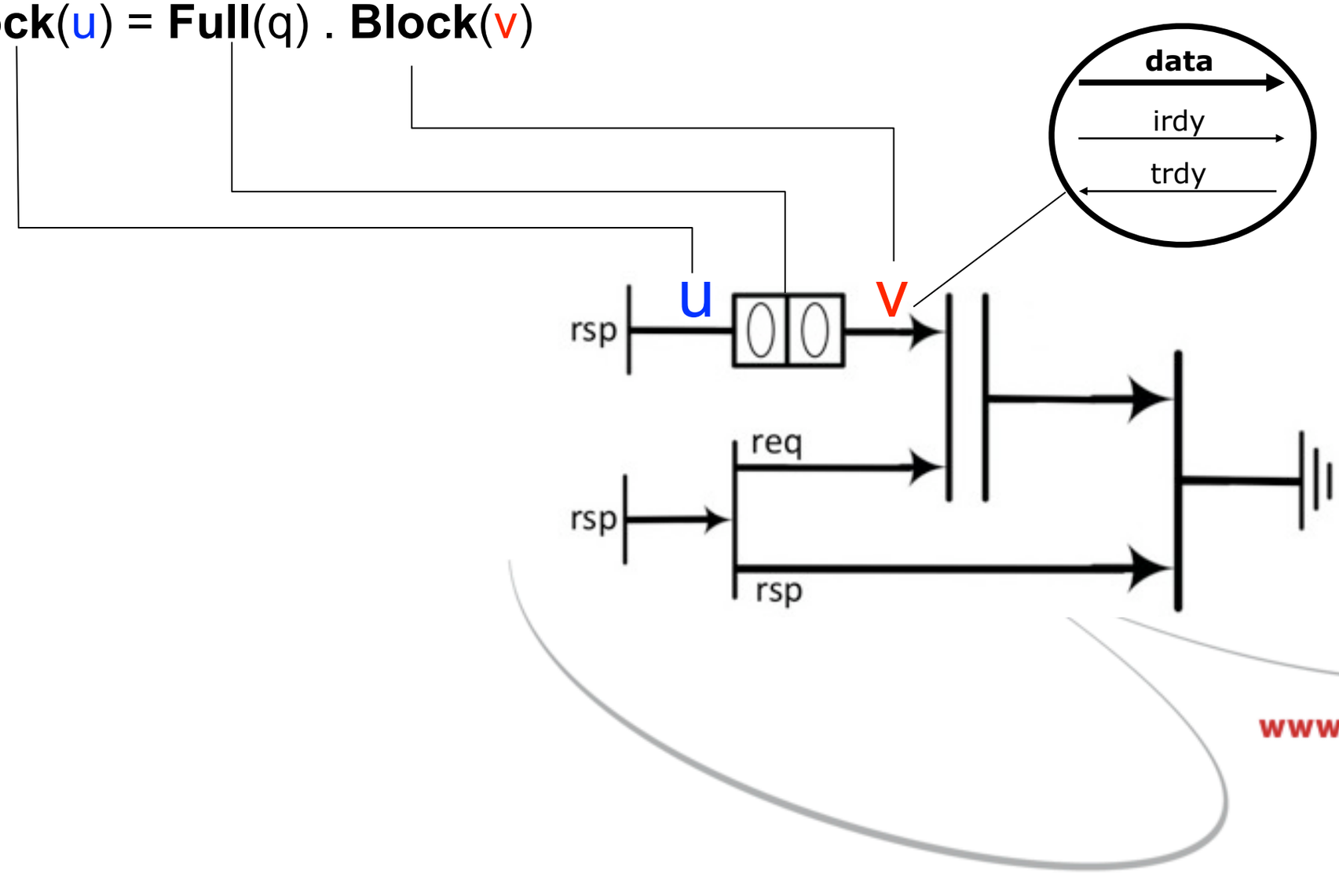
- Depends on the target component connected to the channel
- We look at the input port of the target component



Deadlock equations for queues

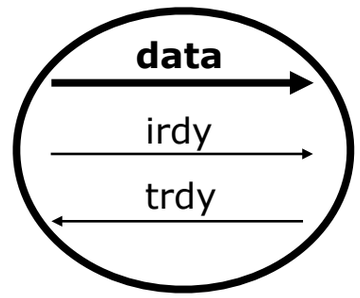
- Queue blocking when full and blocked message at its head
- We look at the input channel of the queue

- **Block**(u) = **Full**(q) . **Block**(v)

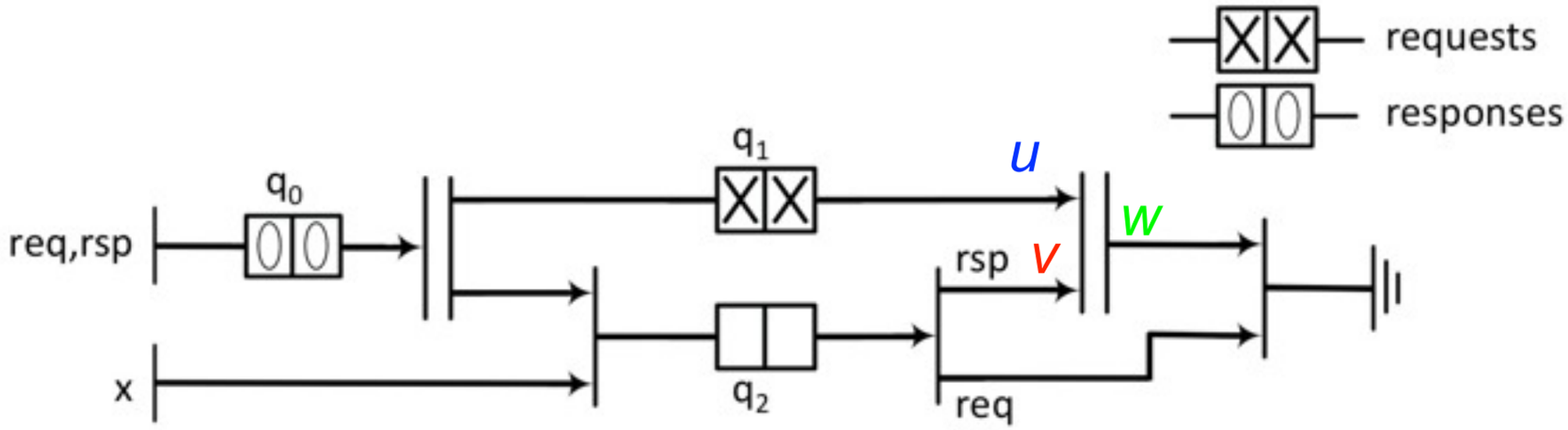


Deadlock equations for a join

- 2 cases
  - output is blocked
  - the other input is idle



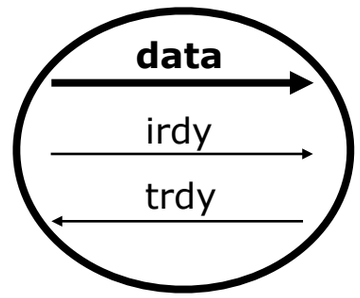
- **Block**(u) = **Idle**(v) + **Block**(w)



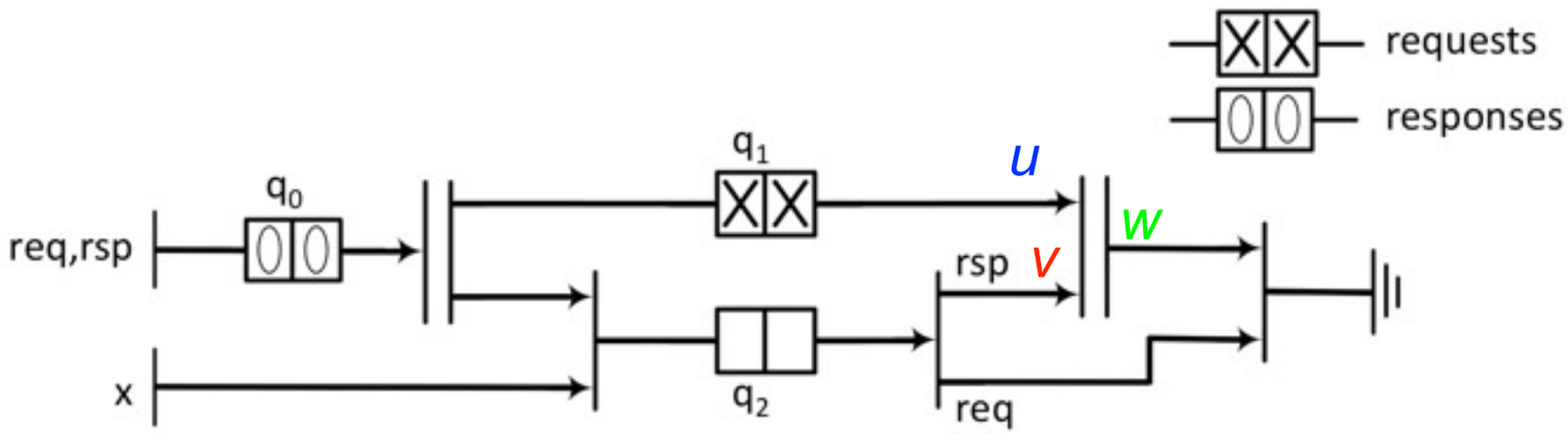
Deadlock equations for a join

- 2 cases
  - output is blocked
  - the other input is idle

We need to know when a channel is idle !



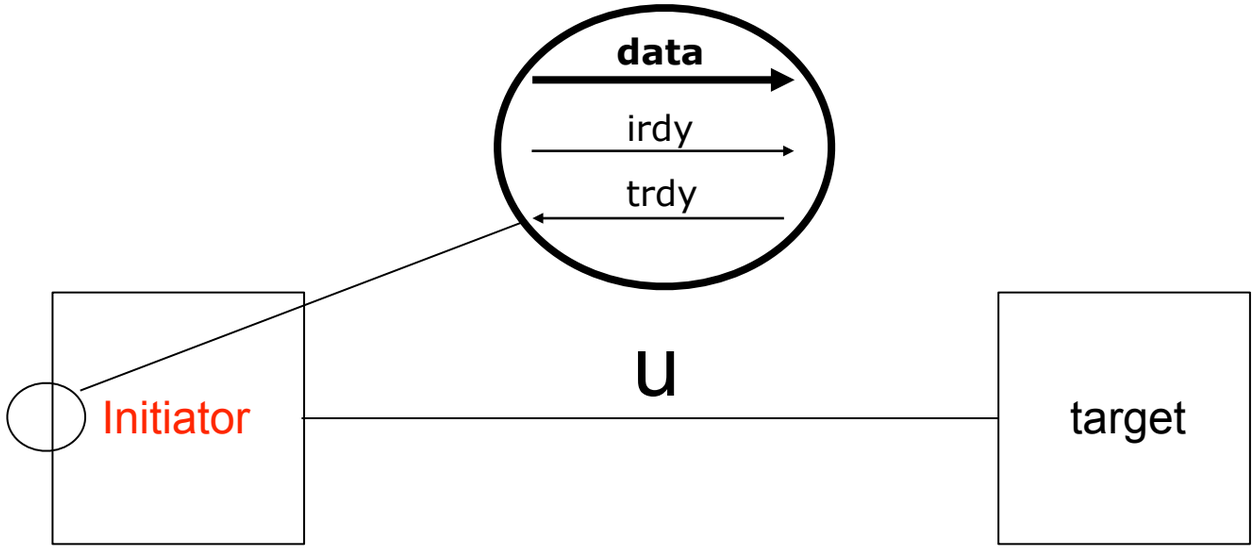
- **Block**(u) = **Idle**(v) + **Block**(w)



www.ou.nl

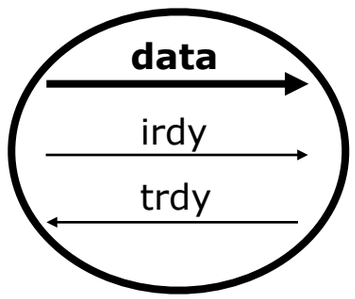
Idle equations for a channel

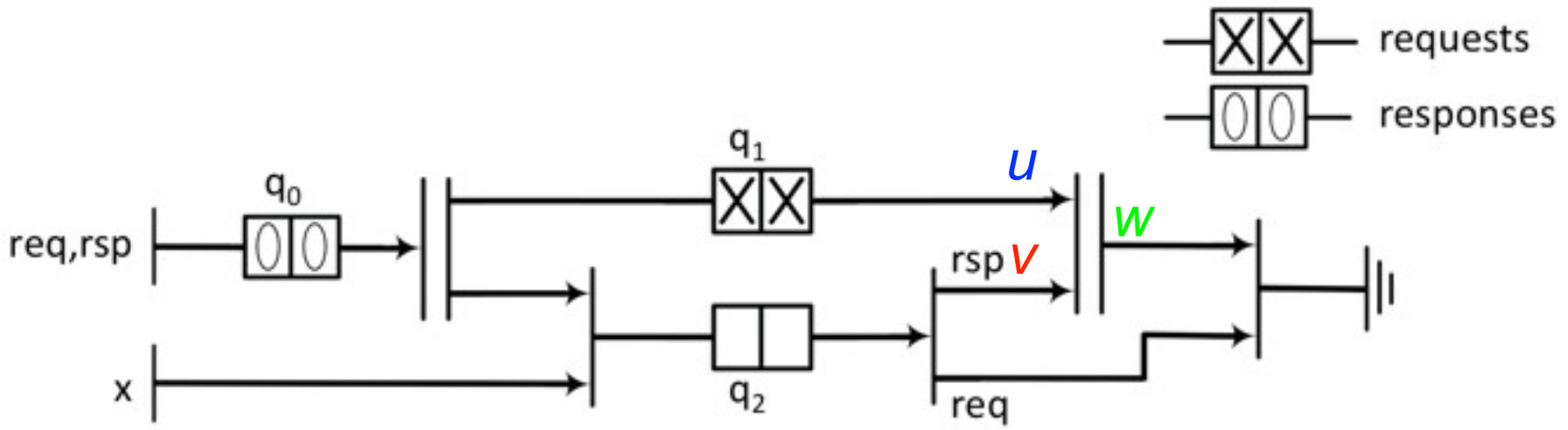
- Depends on the initiator component connected to the channel
- We are looking at the input port of the initiator



Idle equations for a join

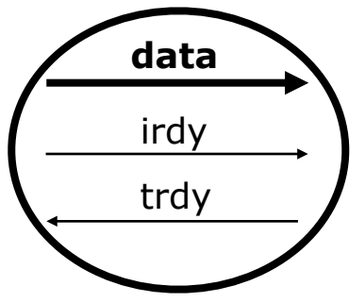
- A join is idle if one of the input channels is idle
- **Idle**(w) = **Idle**(u) + **Idle**(v)

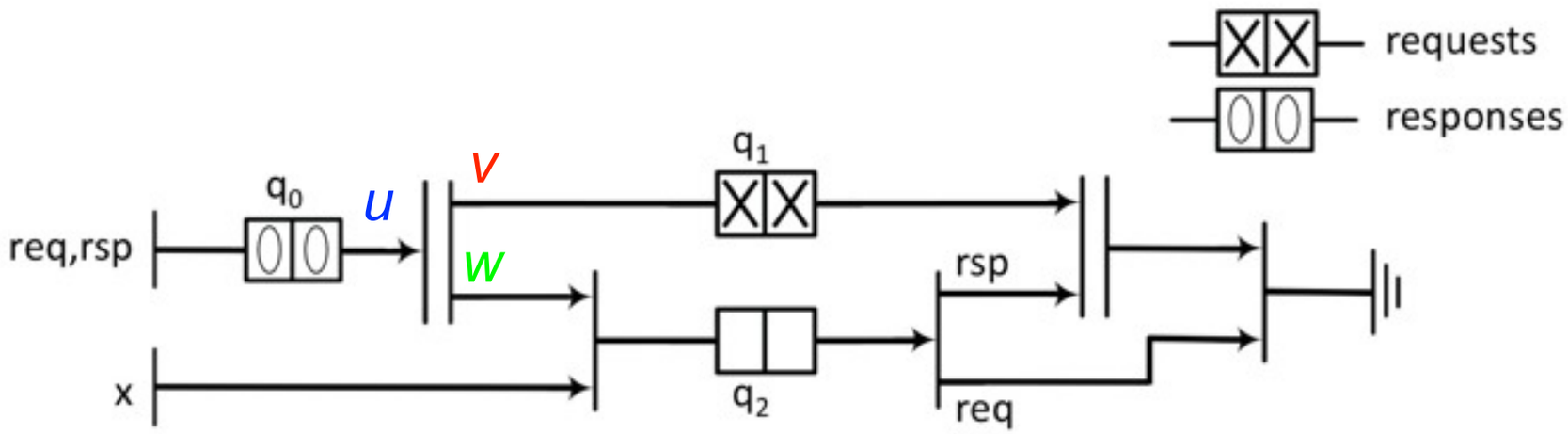




Idle equations for a fork

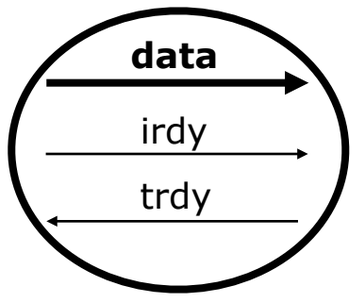
- A fork output is idle if the input is idle or the other output is blocked
- **Idle**(w) = **Idle**(u) + **Block**(v)



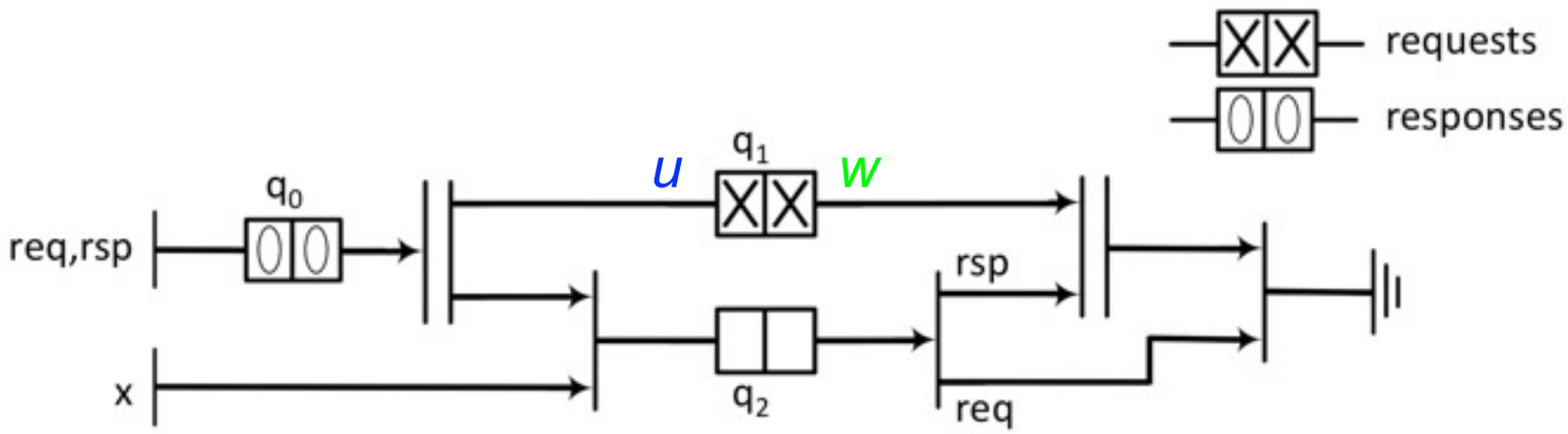


Idle equations for a queue

- A queue is idle if it is empty and its input channel is idle
- This is for one message type which might be blocked by another type



- **Idle**(w) = **Empty**(q) . **Idle**(u) + **Block**(w')
  - where w' is a message with a type different from w



Our quest for "dead" queues

- Definition of a deadlock
  - F (u.irdy => G~u.trdy)

- We look for a "dead" queue
  - with a message in it (u.irdy)
  - output blocked (G~u.trdy)

- Over approximation
  - configuration not always reachable
  - we may output false deadlocks

General approach for deadlock detection in xMAS networks

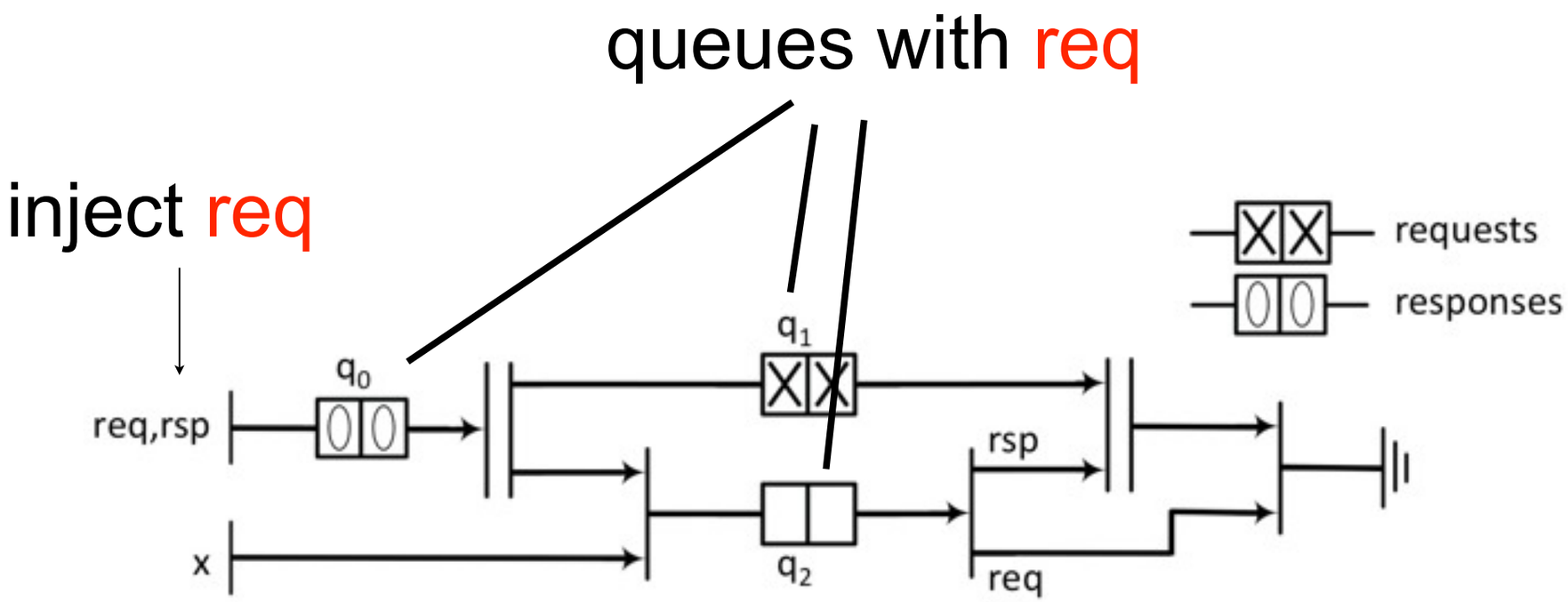
- Define deadlock equations for all components
  - Equations capture the reason why a component is idle or blocking

- Build a labelled waiting graph for each queue
  - Labels correspond to the equations
  - Graph captures the topology, i.e., the dependencies between the xMAS components

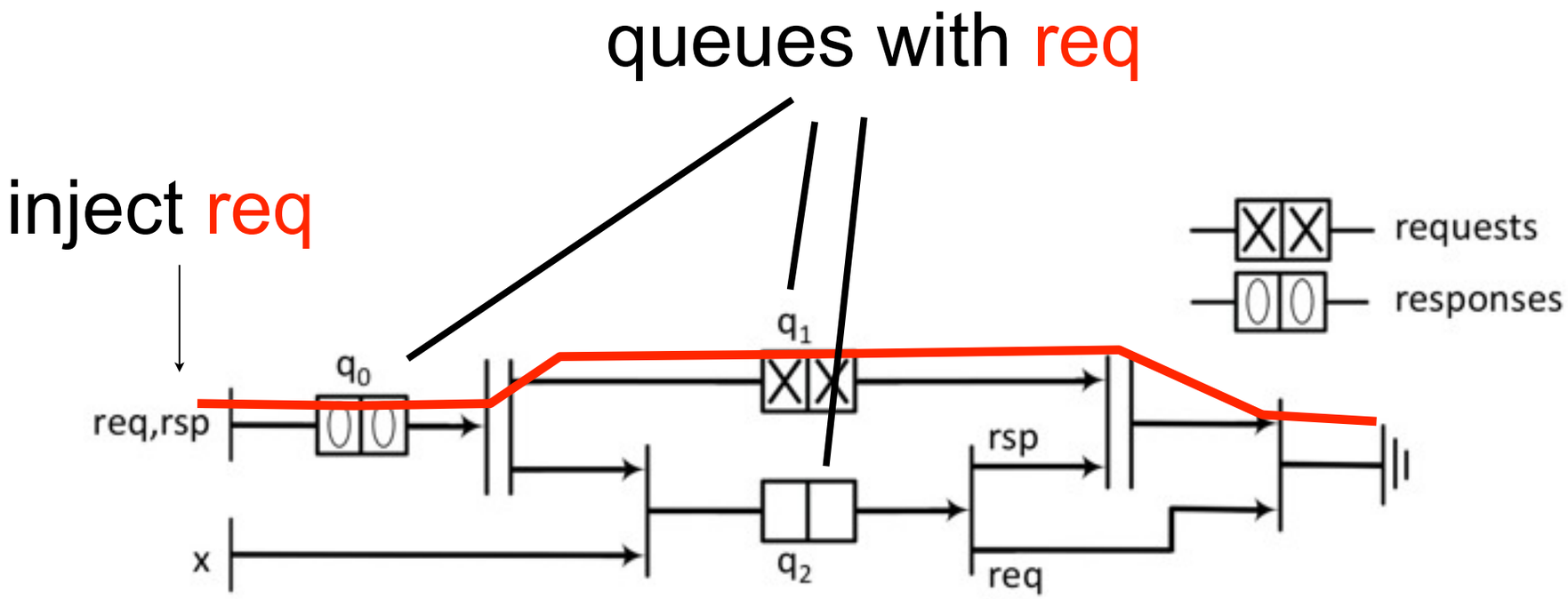
- Search for a feasible logically closed subgraph
  - Corresponds to a deadlock situation
  - Feasibility checked using Linear Programming

- This approach may output unreachable deadlocks
  - A first step generates invariants to rule out false deadlocks
  - Invariants are rather weak and simple - false deadlocks are in theory still possible

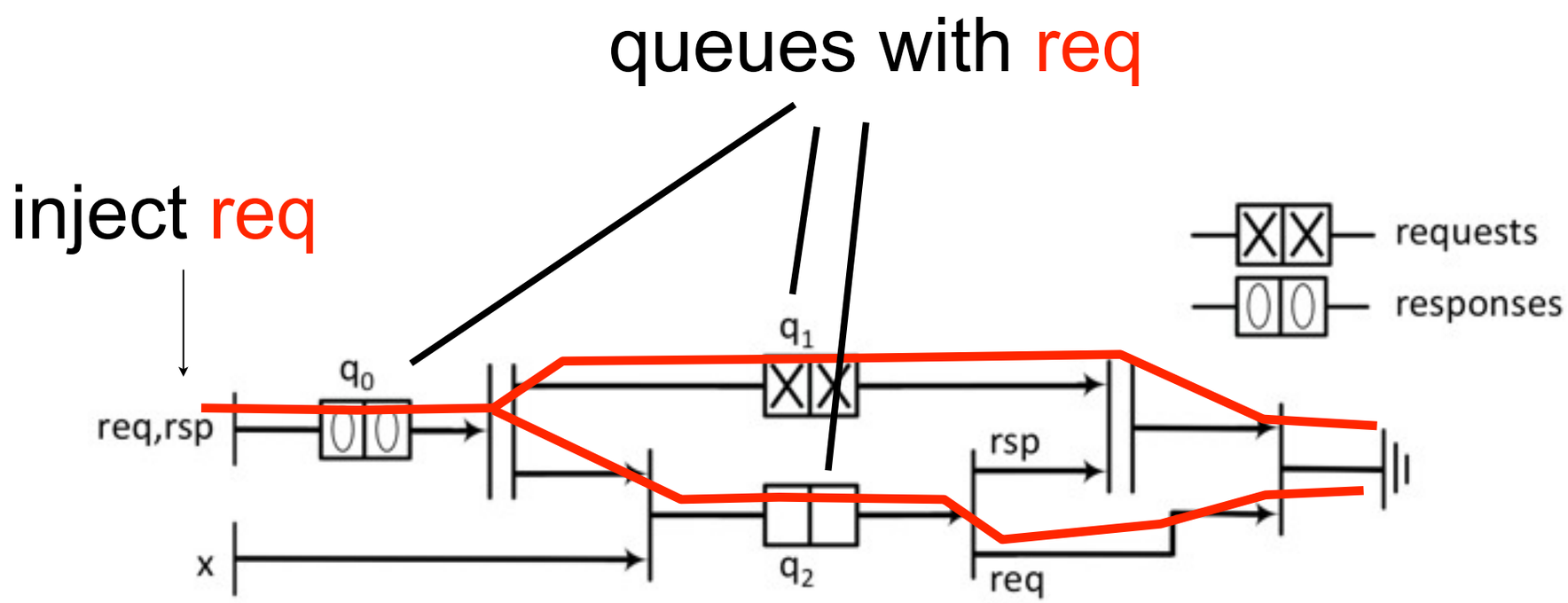
Step 1 / simulation - req



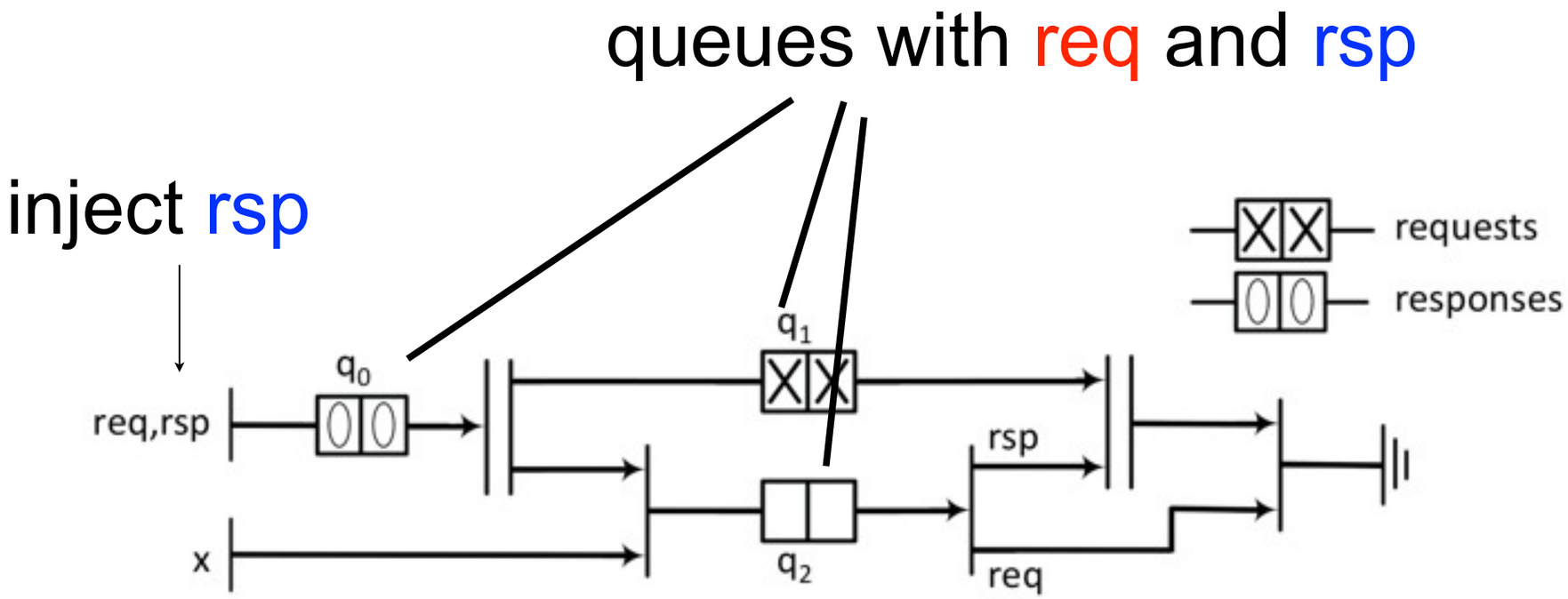
Step 1 / simulation - req



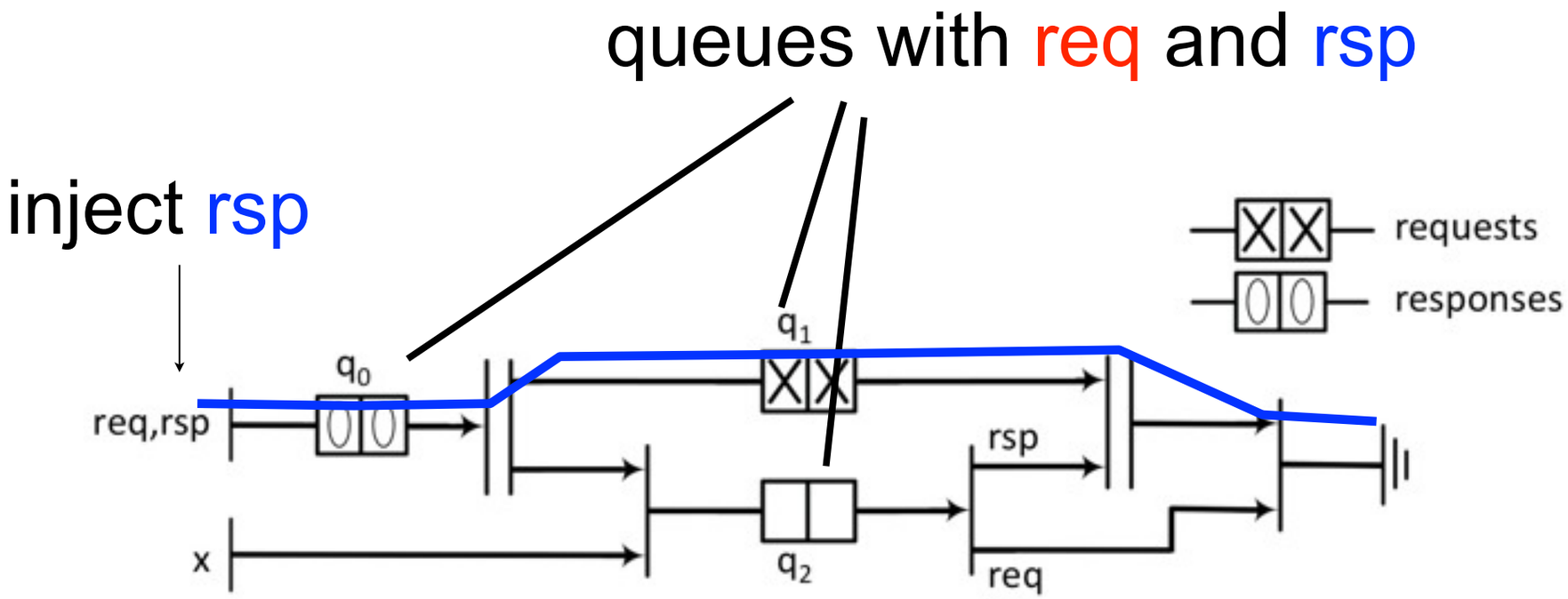
Step 1 / simulation - req



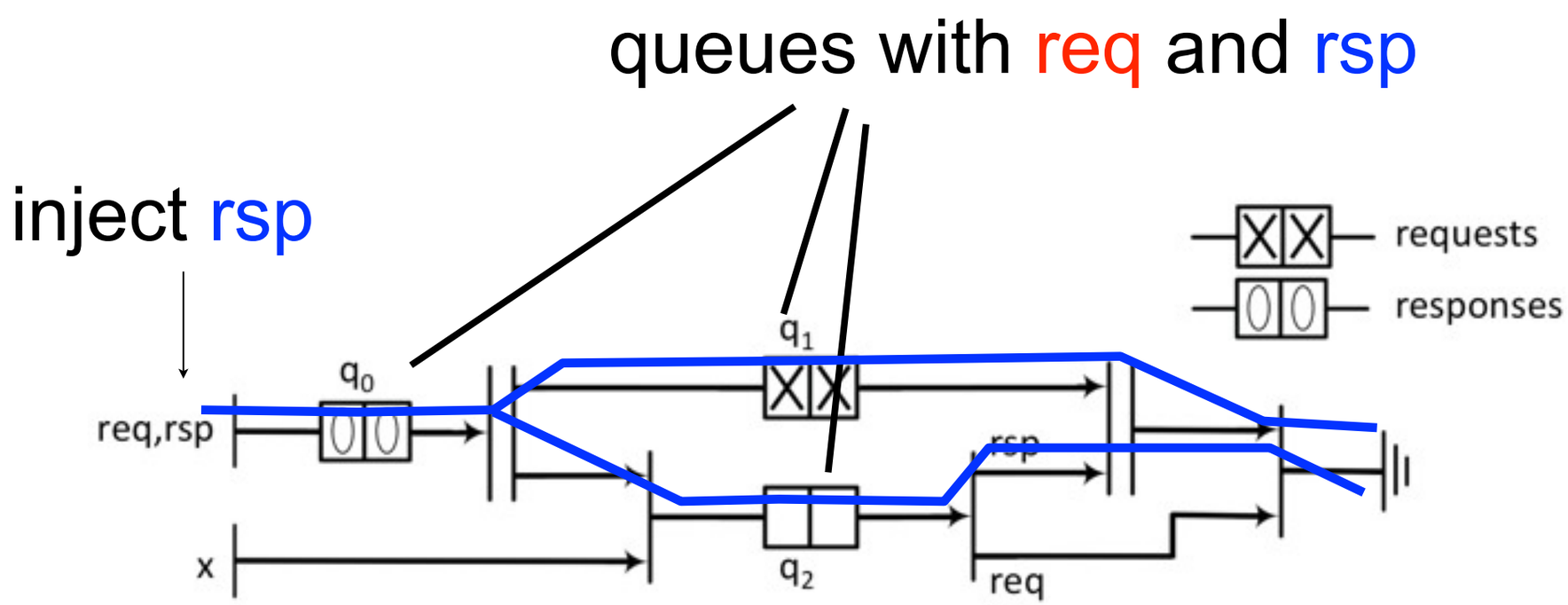
Step 1 / simulation - rsp



Step 1 / simulation - rsp



Step 1 / simulation - rsp



General approach for deadlock detection in xMAS networks

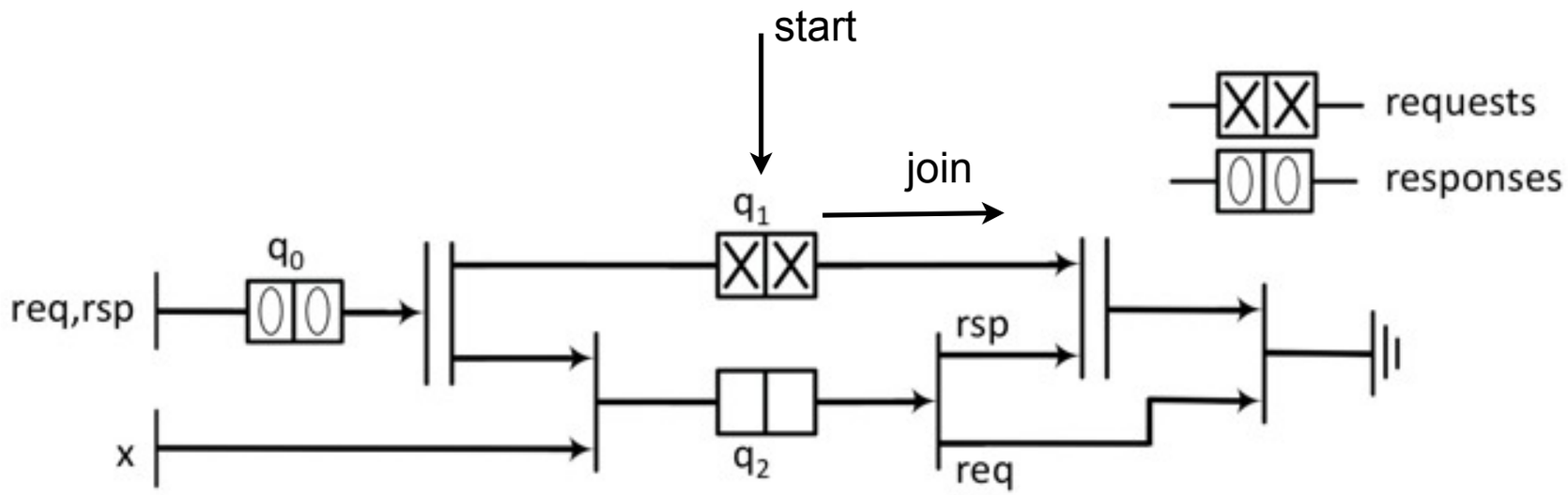
- Define deadlock equations for all components
  - Equations capture the reason why a component is idle or blocking

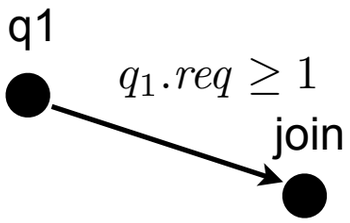
- Build a labelled waiting graph for each queue
  - Labels correspond to the equations
  - Graph captures the topology, i.e., the dependencies between the xMAS components

- Search for a feasible logically closed subgraph
  - Corresponds to a deadlock situation
  - Feasibility checked using Linear Programming

- This approach may output unreachable deadlocks
  - A first step generates invariants to rule out false deadlocks
  - Invariants are rather weak and simple - false deadlocks are in theory still possible

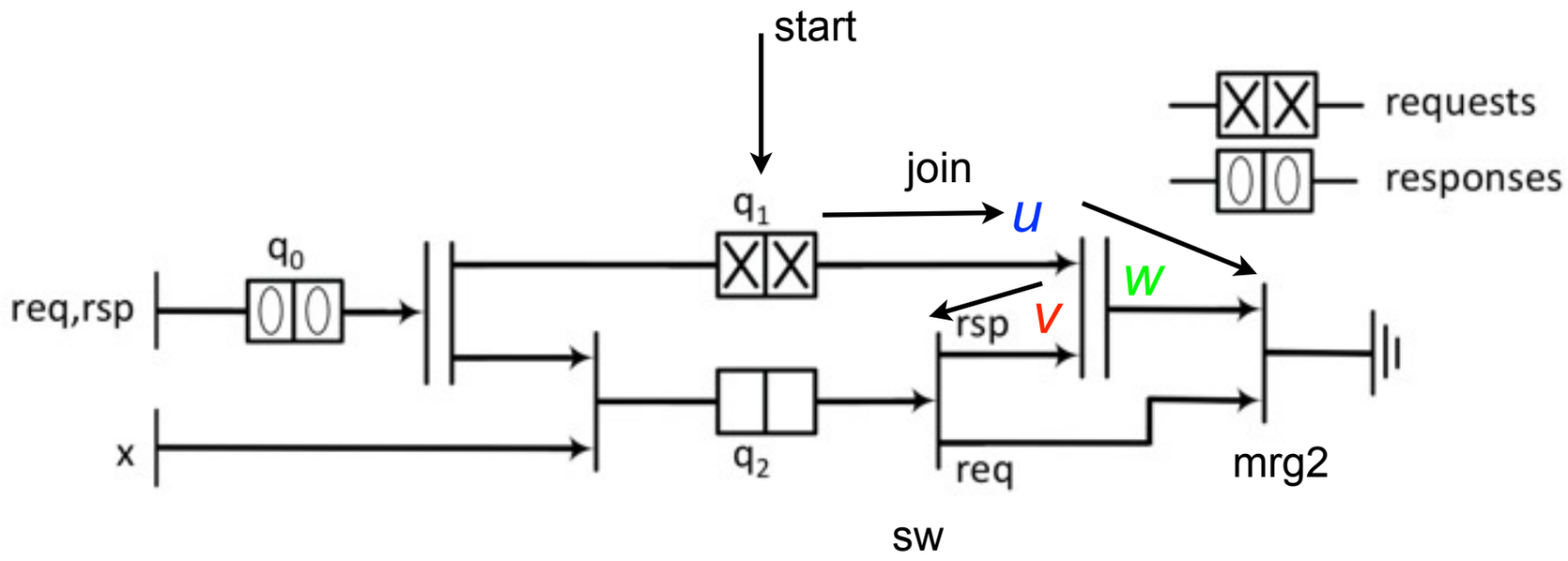
Step 2 / labelled dependency graph (1)



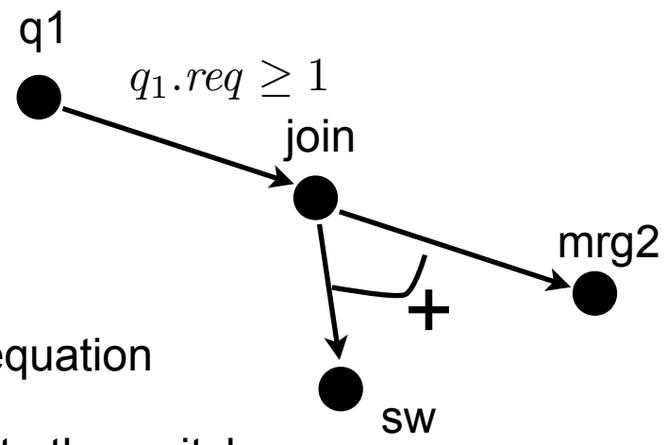


start with a message in q1 and visit the join

Step 2 / labelled dependency graph (2)



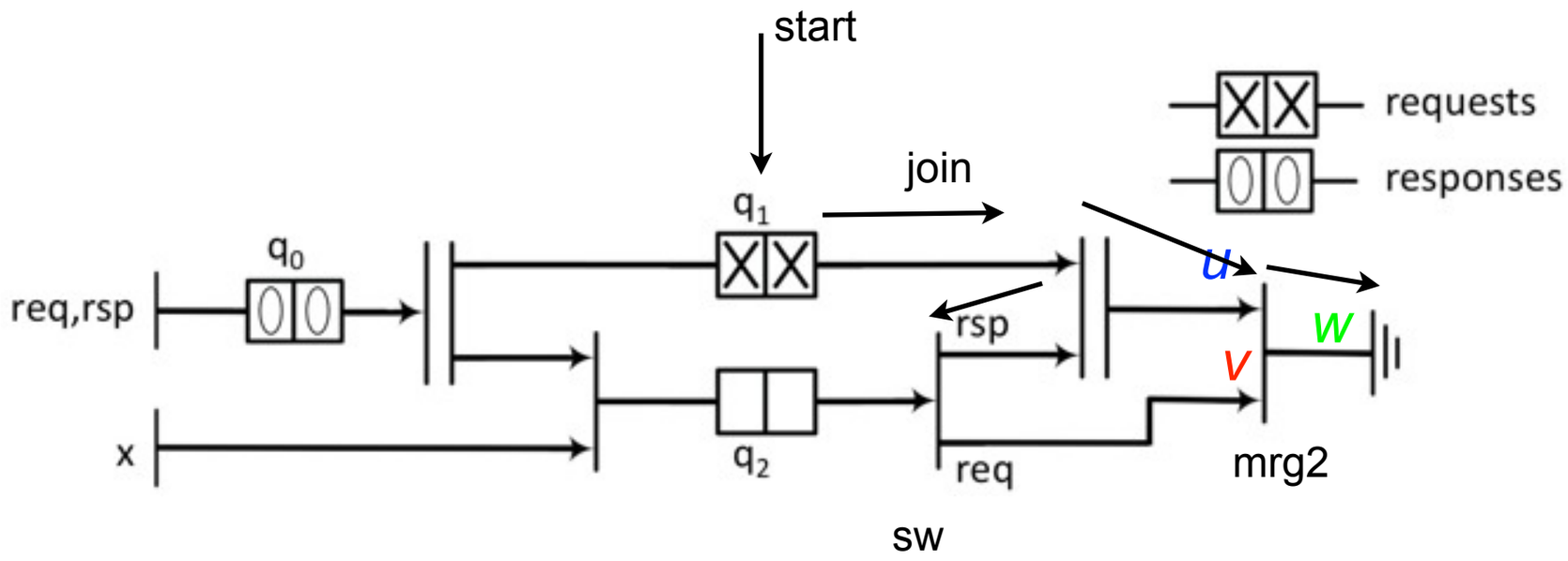
Block(u) = **Idle**(v) + **Block**(w)



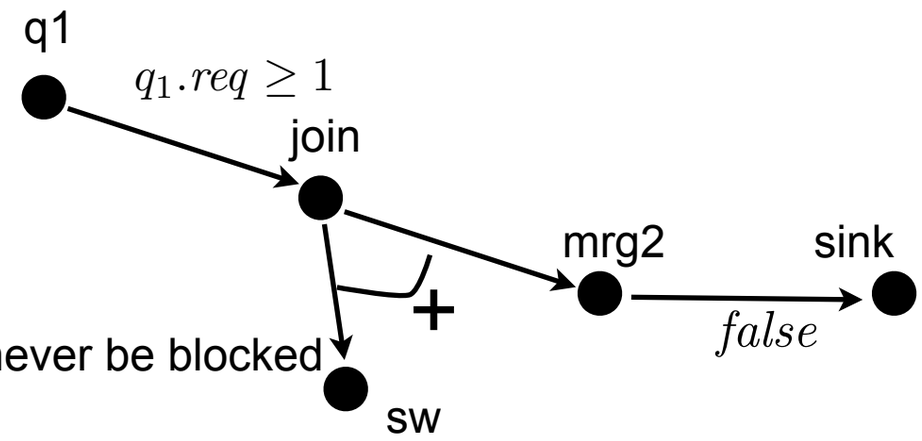
analyse the join according to its deadlock equation

we go forward to the merge and backward to the switch

Step 2 / labelled dependency graph (2)



Block(u) = **Block**(w)

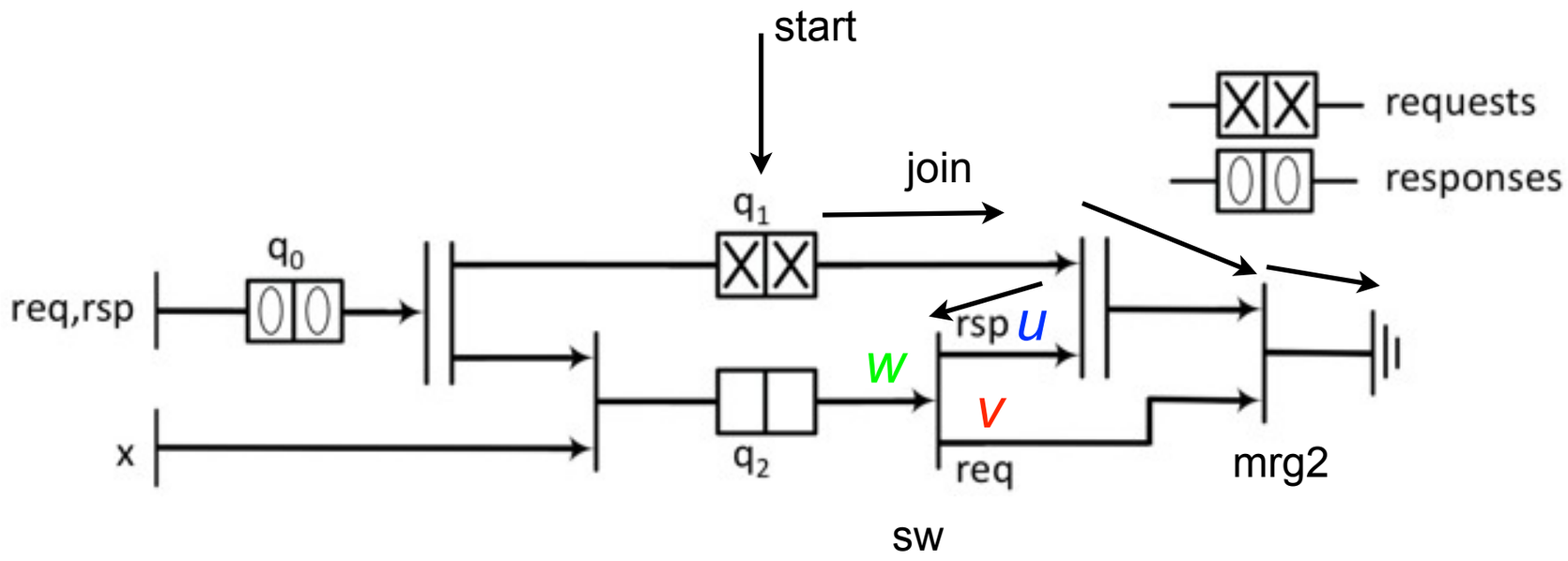


forwards to the switch - then the sink can never be blocked

we assume fair sinks

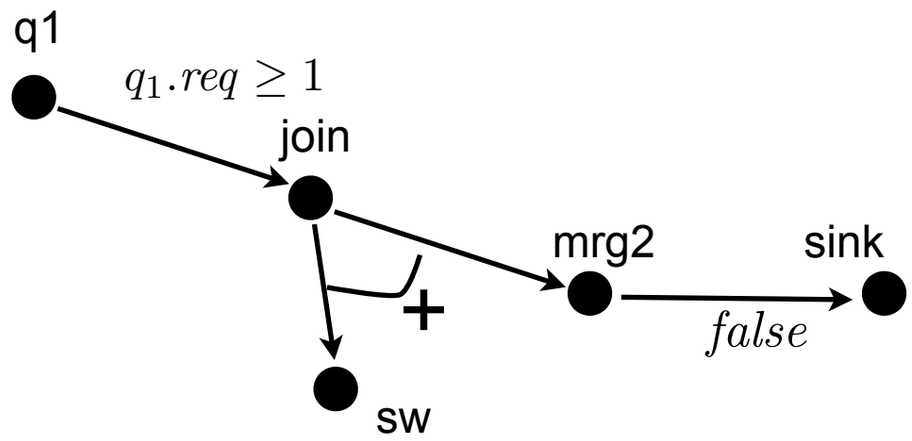
www.ou.nl

Step 2 / labelled dependency graph (2)

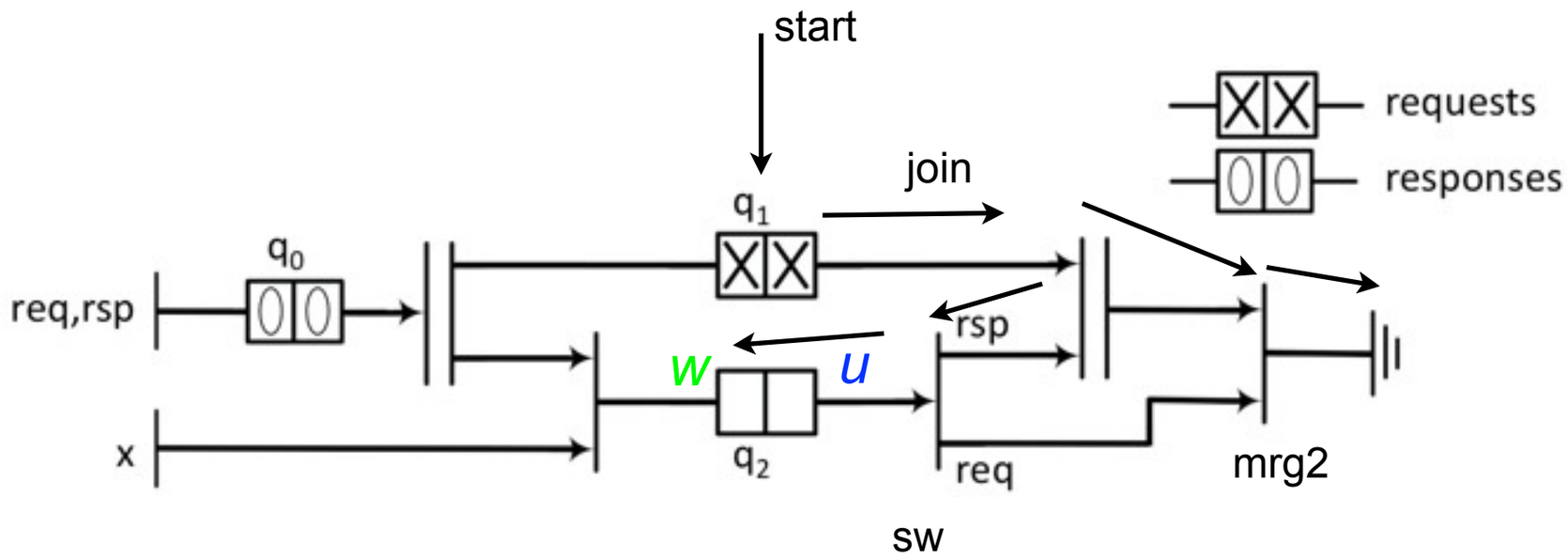


Idle(u) = **Idle**(w)

backwards to the switch



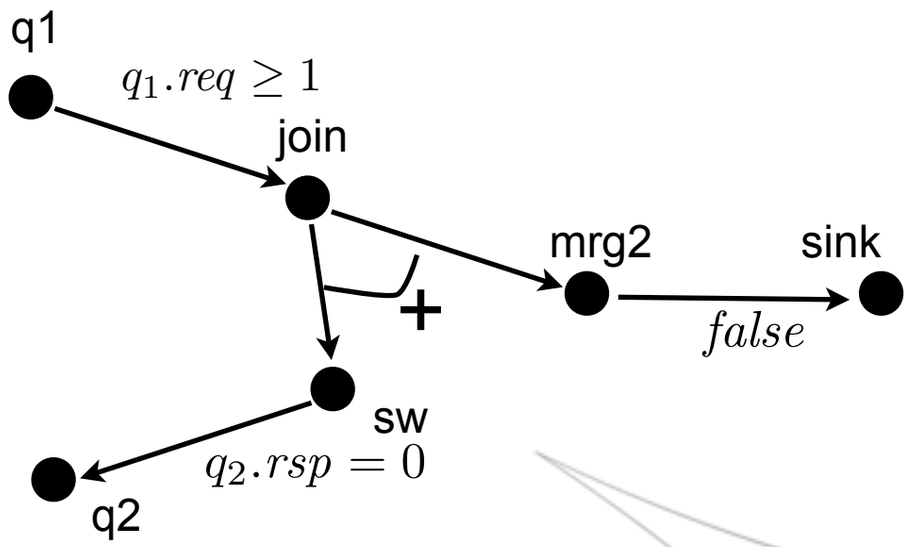
Step 2 / labelled dependency graph (2)



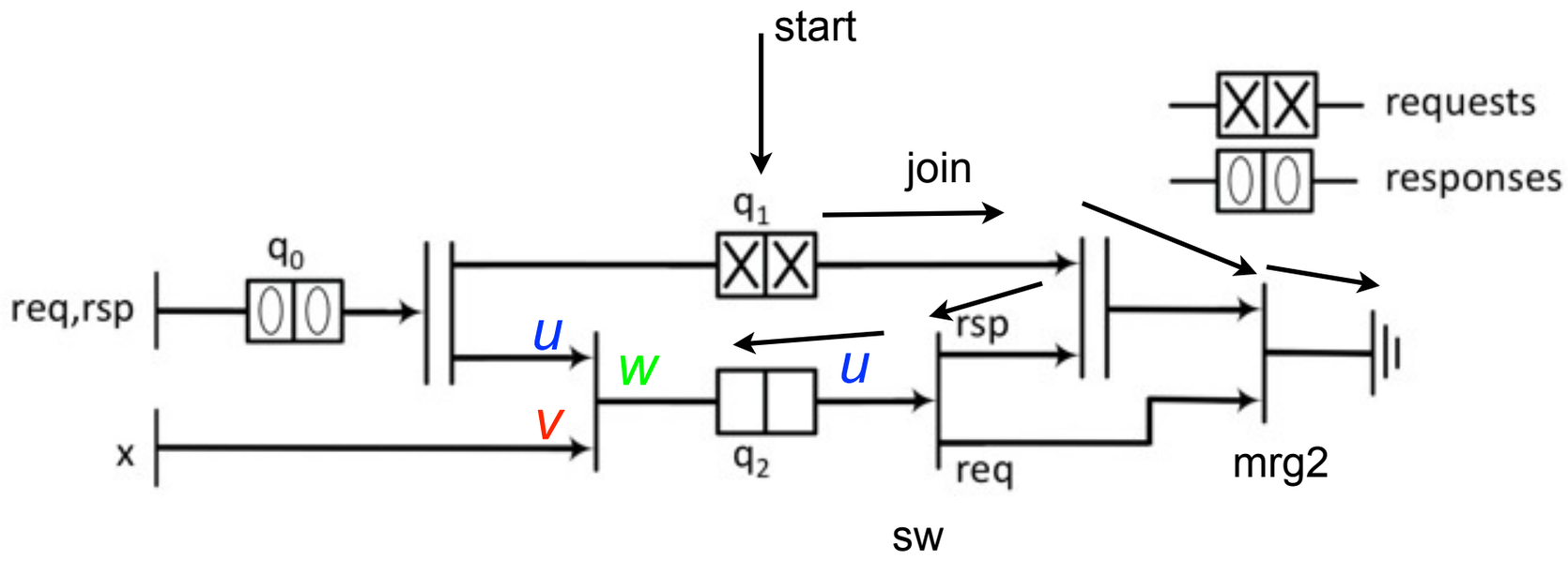
Idle(u) = **Idle**(w) . **Empty**(q2)

backwards to the queue

note that we forgot the **Block**(w') case



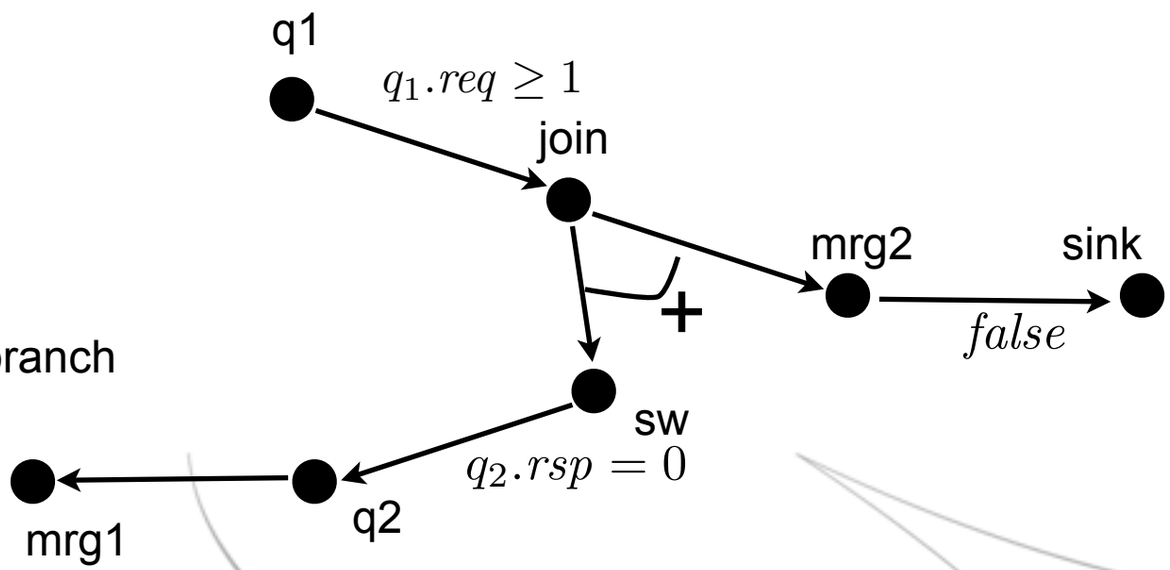
Step 2 / labelled dependency graph (2)



Idle(w) = **Idle**(u) + **Idle**(v)

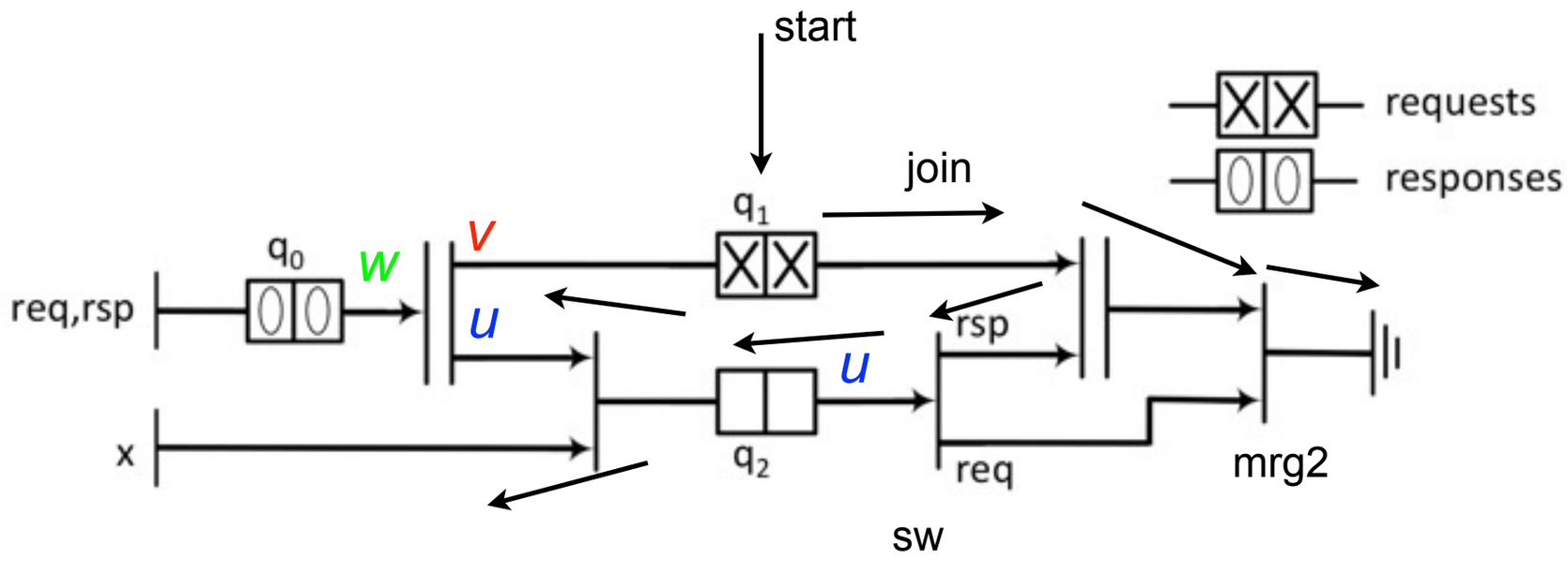
backwards to the merge and branch

note branching is bad for us



www.ou.nl

Step 2 / labelled dependency graph (2)

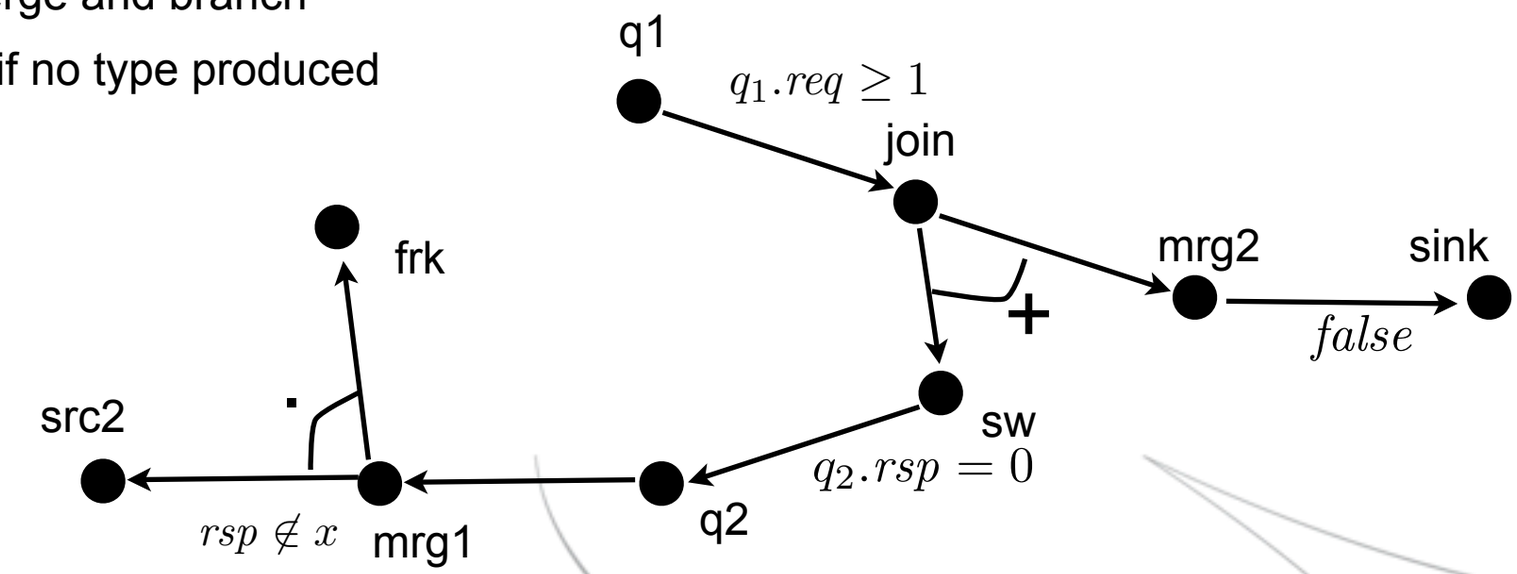


Idle(u) = **Block**(v) + **Idle**(w)

backwards to the merge and branch

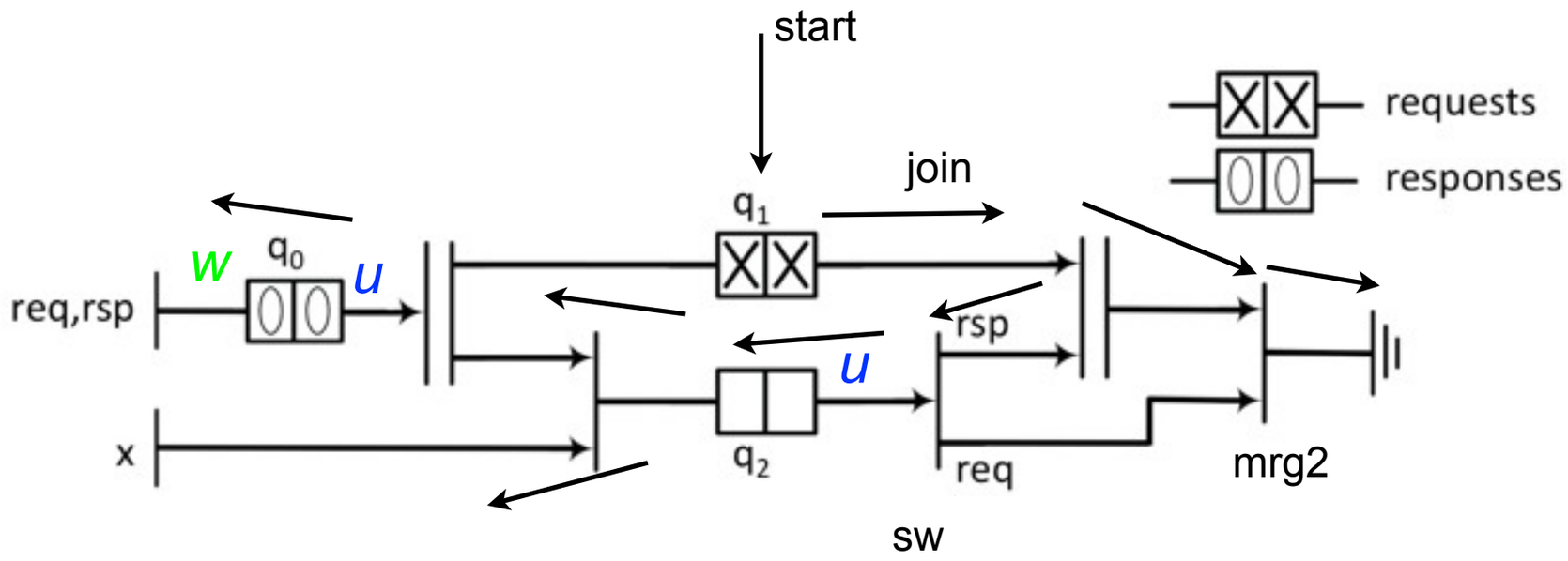
to the source - idle if no type produced

to the fork



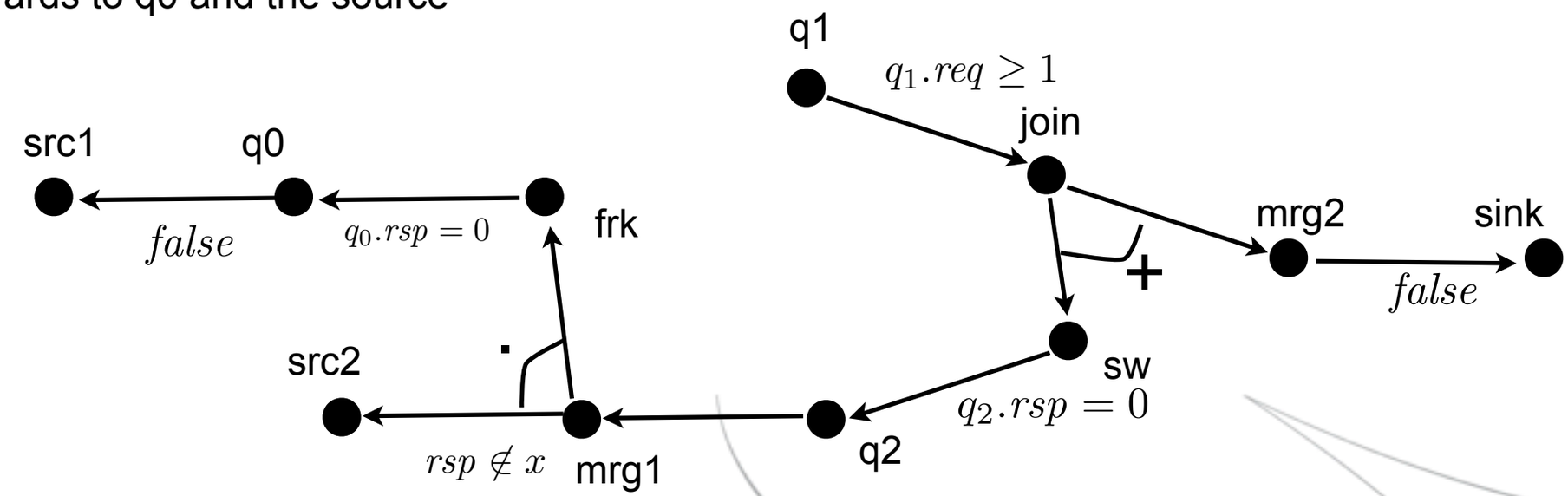
www.ou.nl

Step 2 / labelled dependency graph (2)

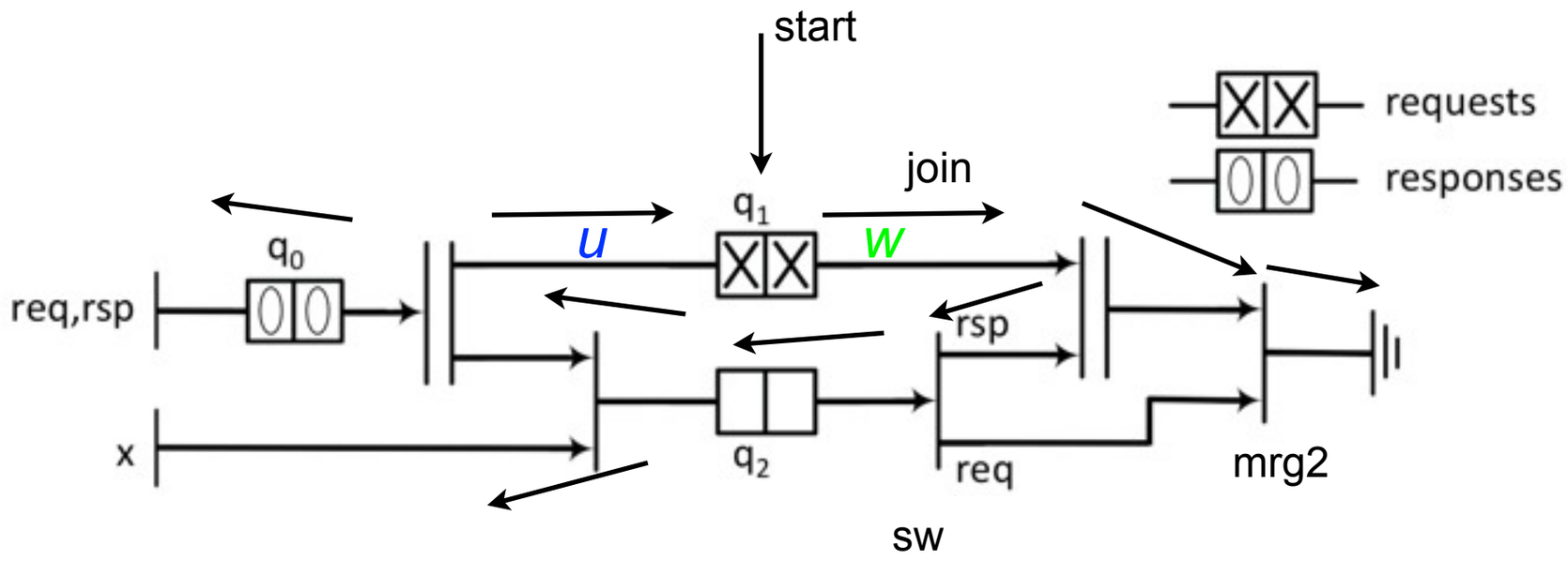


Idle(u) = **Idle**(w) . **Empty**(q0)

backwards to q0 and the source

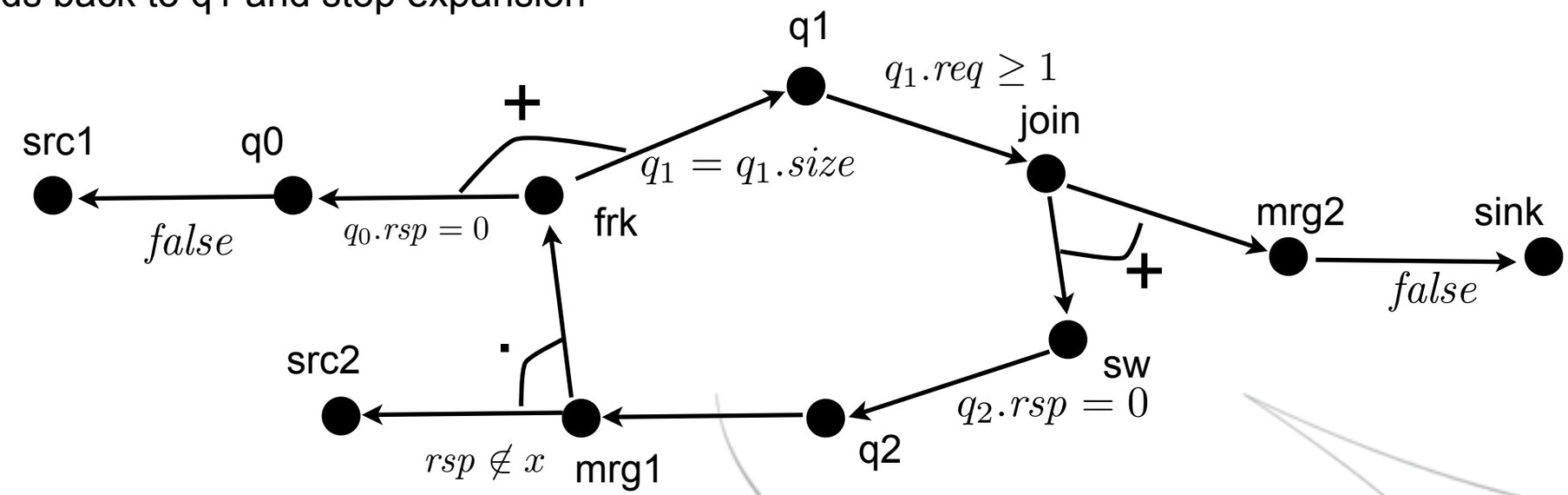


Step 2 / labelled dependency graph (2)



Block(u) = **Block**(w) . **Full**(q1)

forwards back to q1 and stop expansion



General approach for deadlock detection in xMAS networks

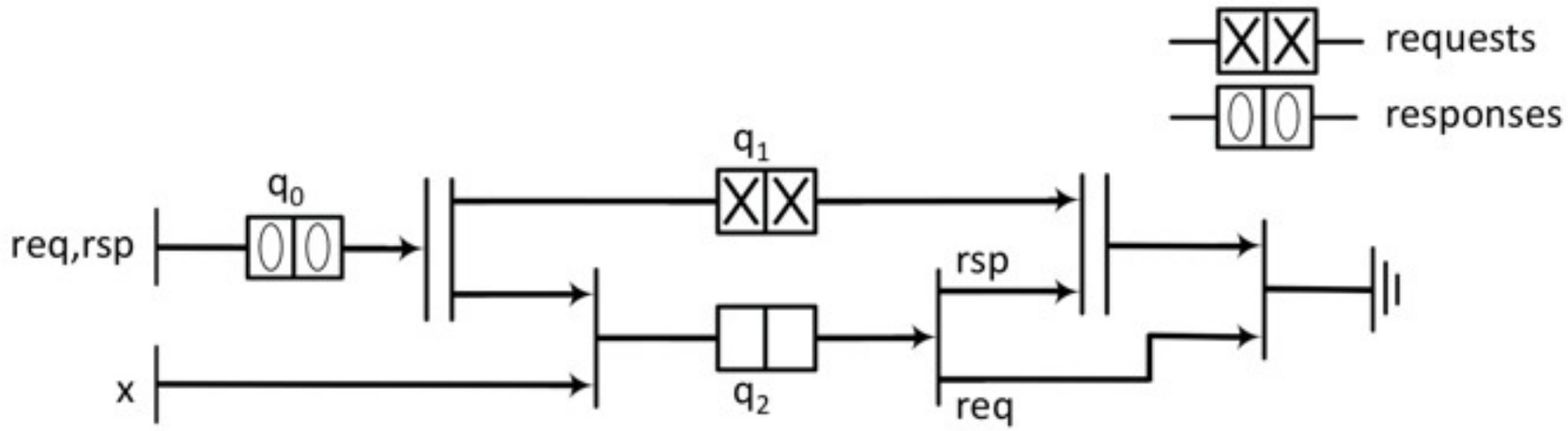
- Define deadlock equations for all components
  - Equations capture the reason why a component is idle or blocking

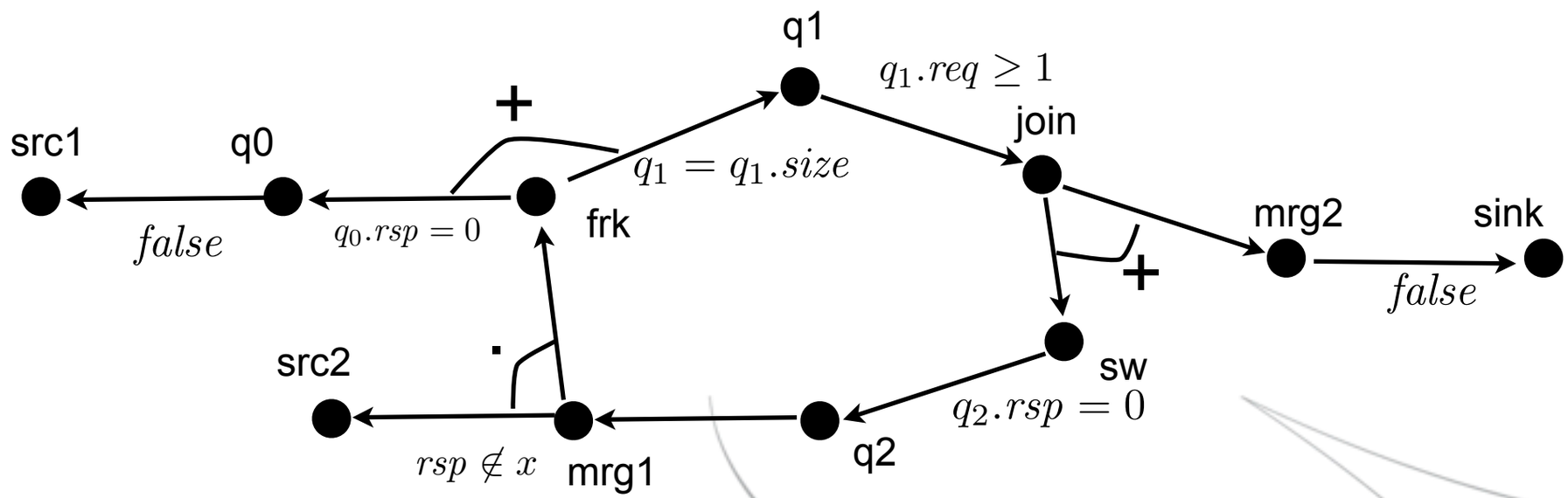
- Build a labelled waiting graph for each queue
  - Labels correspond to the equations
  - Graph captures the topology, i.e., the dependencies between the xMAS components

- Search for a feasible logically closed subgraph
  - Corresponds to a deadlock situation
  - Feasibility checked using Linear Programming

- This approach may output unreachable deadlocks
  - A first step generates invariants to rule out false deadlocks
  - Invariants are rather weak and simple - false deadlocks are in theory still possible

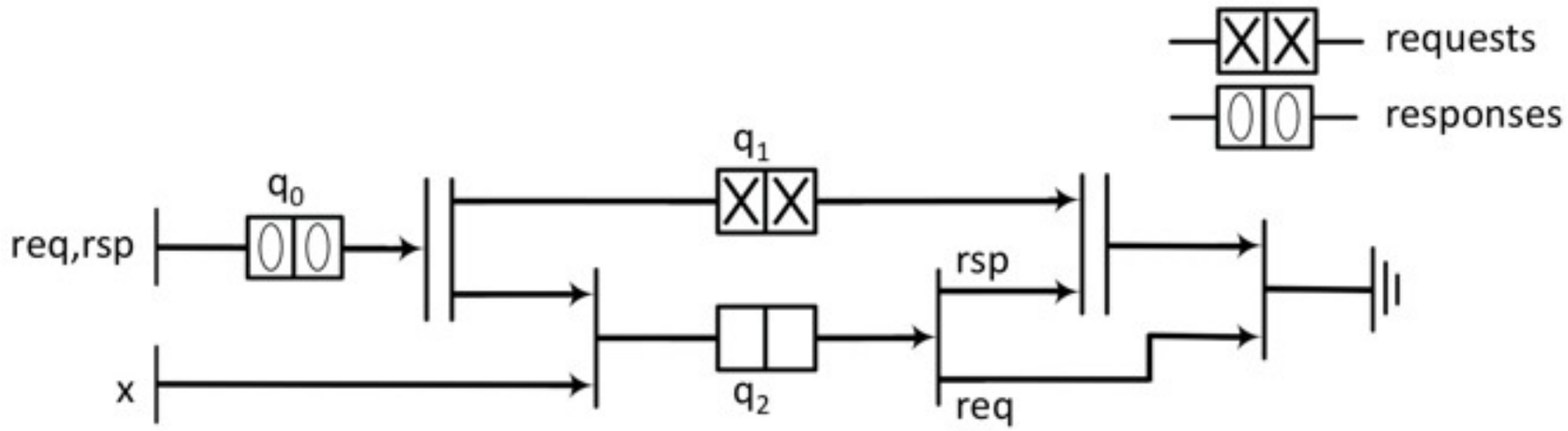
Step 2 / logically closed subgraph 1

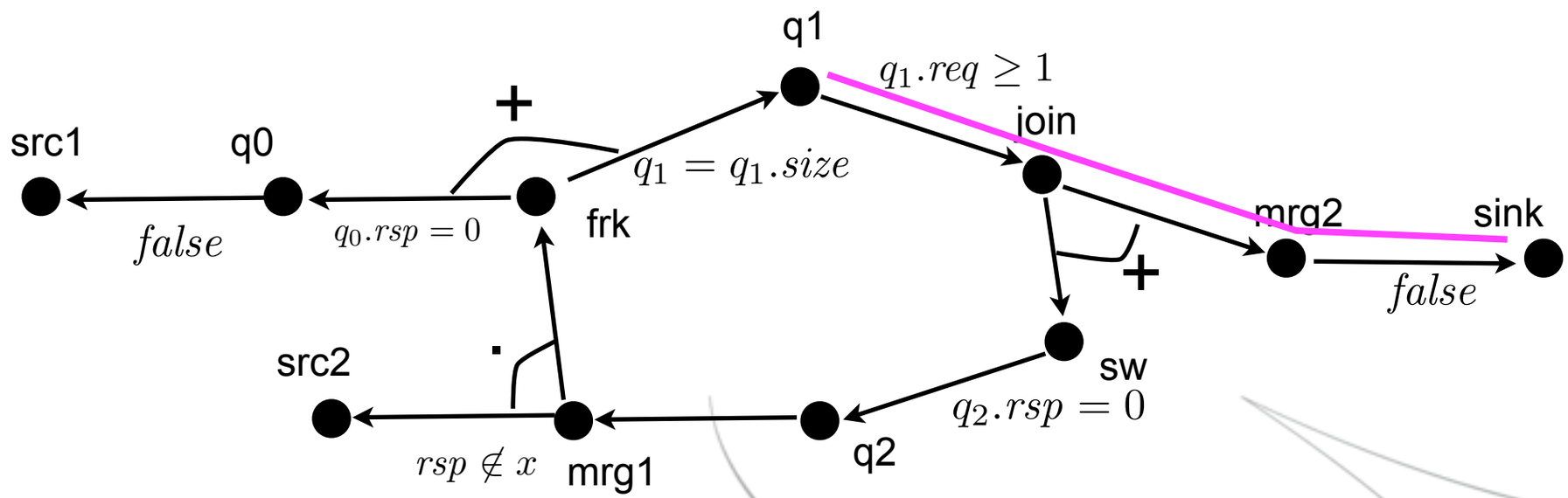




www.ou.nl

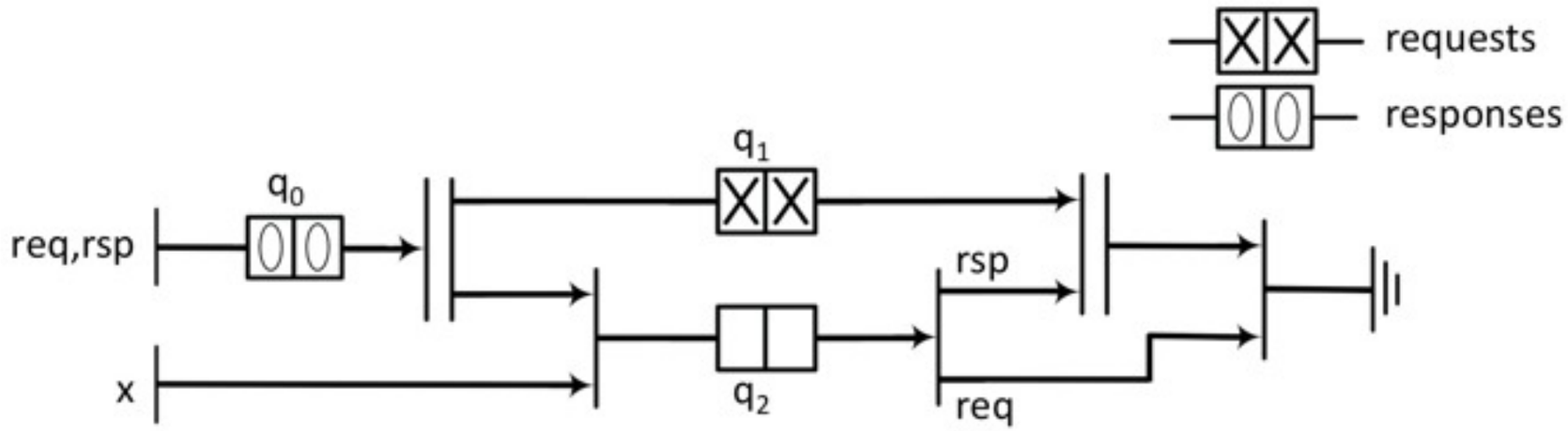
Step 2 / logically closed subgraph 1

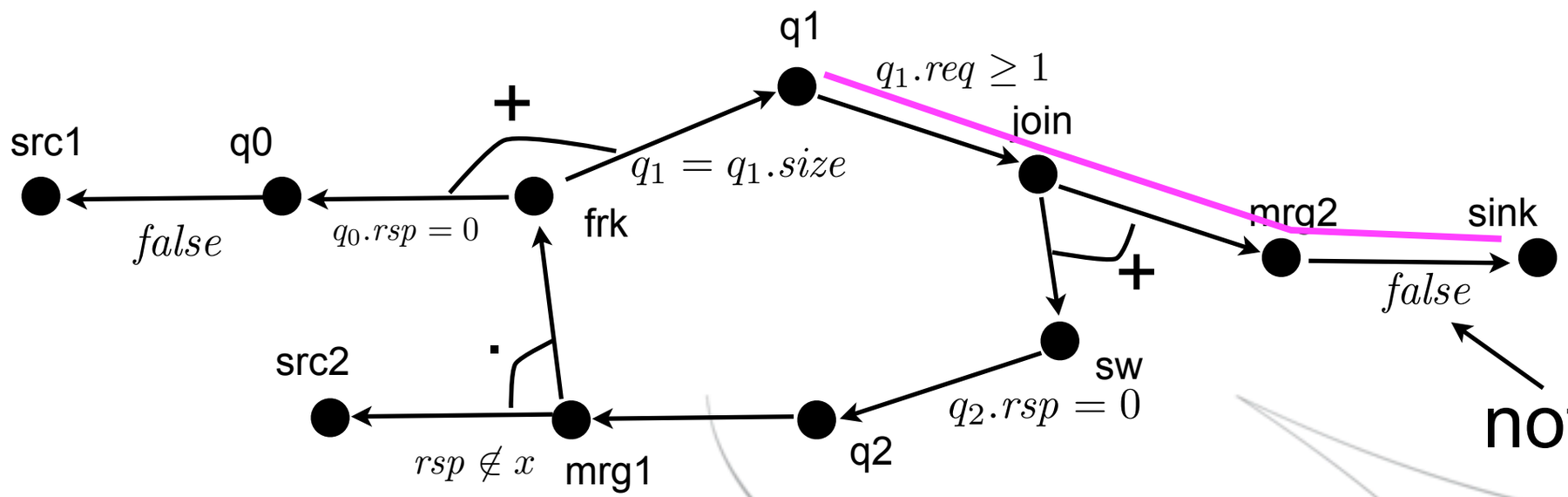




www.ou.nl

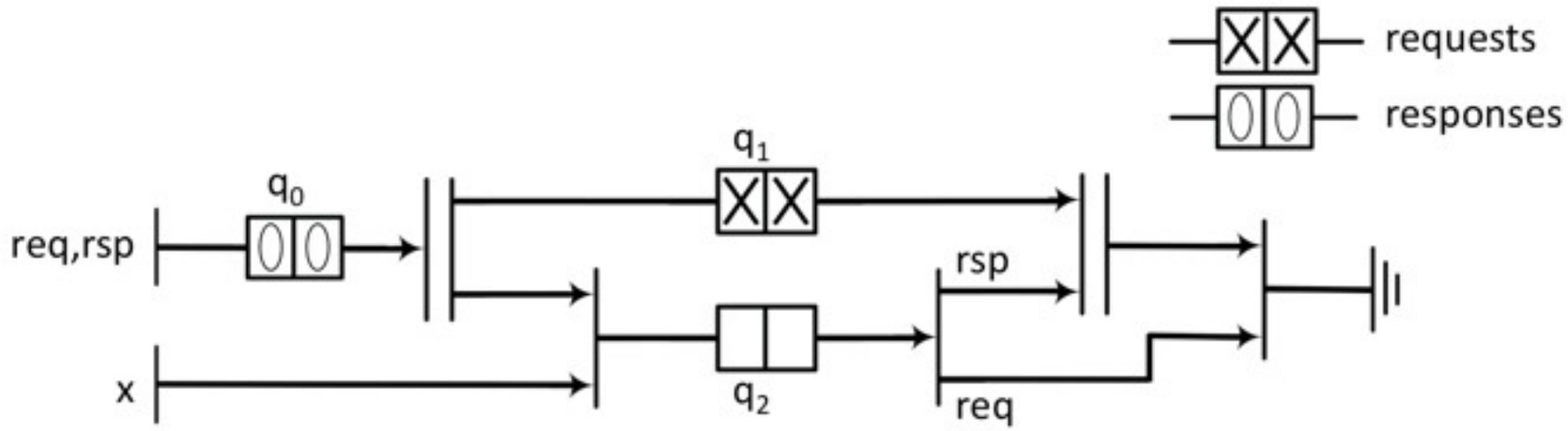
Step 2 / logically closed subgraph 1

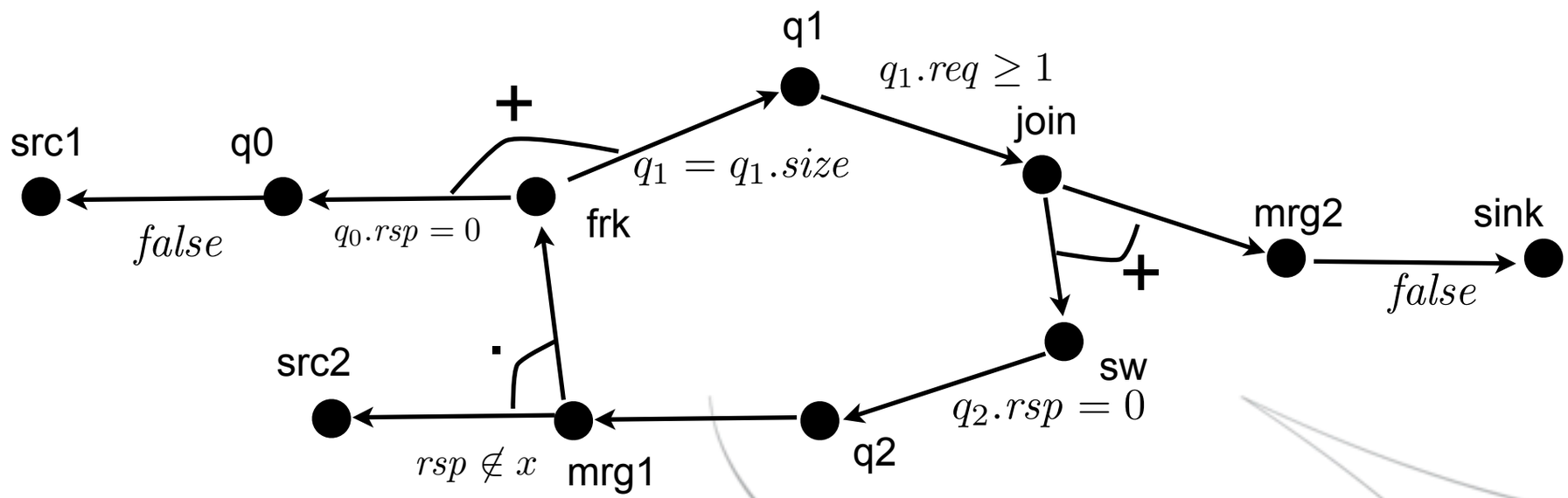




not feasible

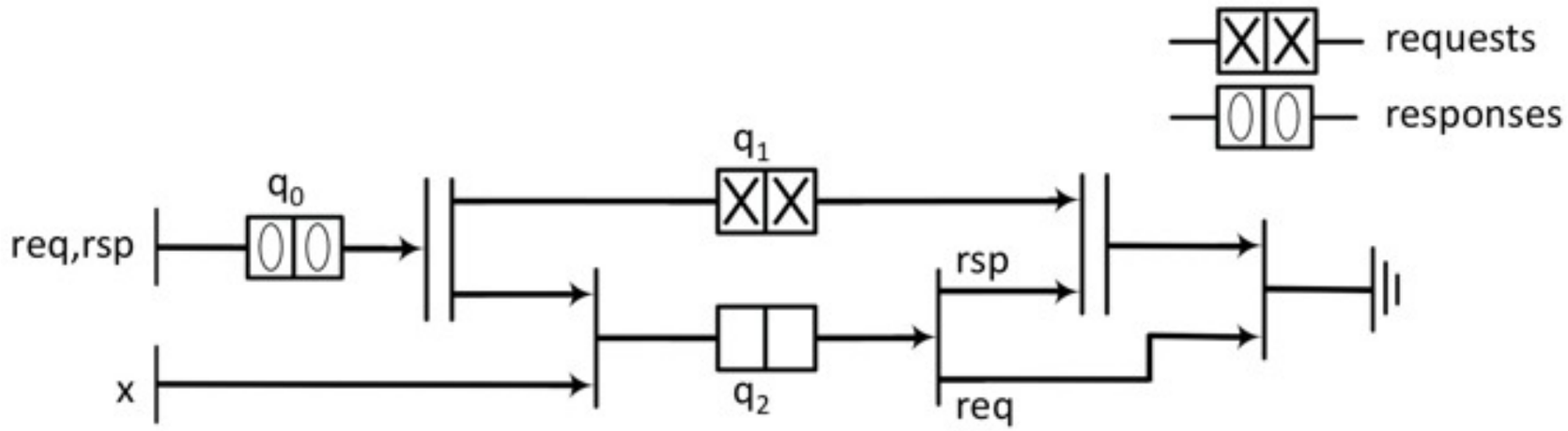
Step 2 / logically closed subgraph 2

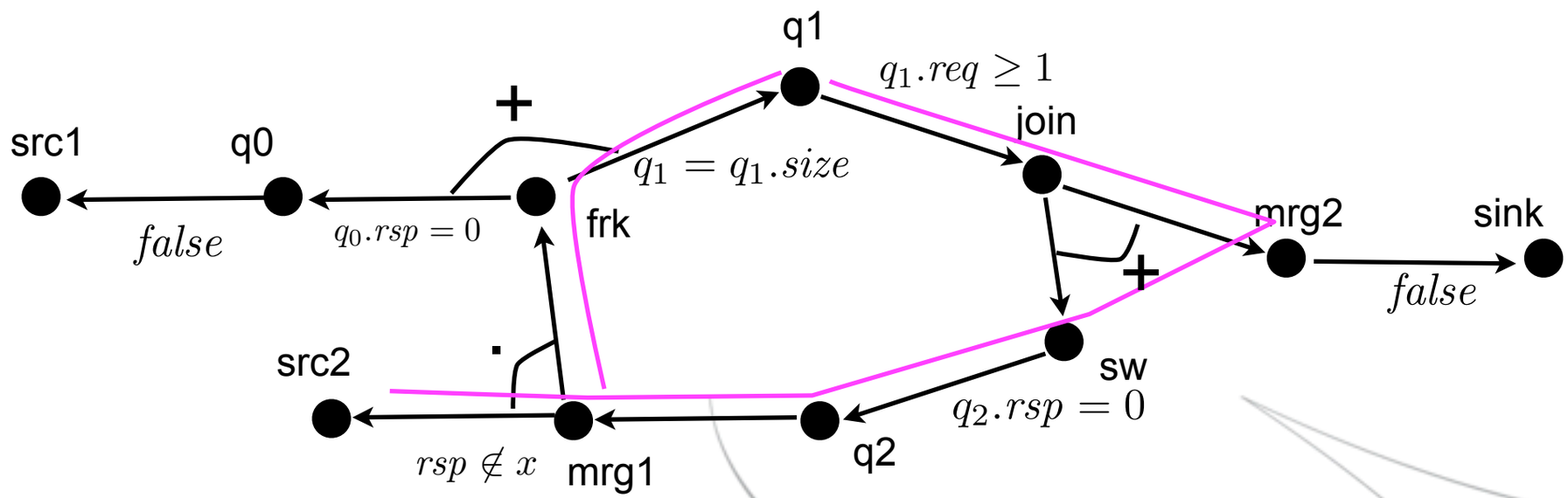




www.ou.nl

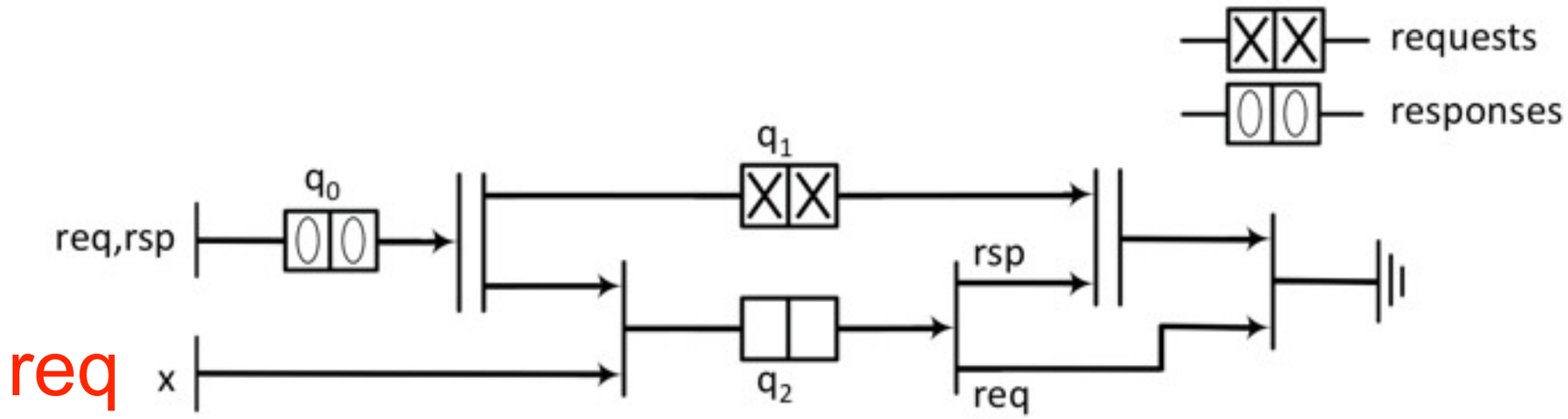
Step 2 / logically closed subgraph 2

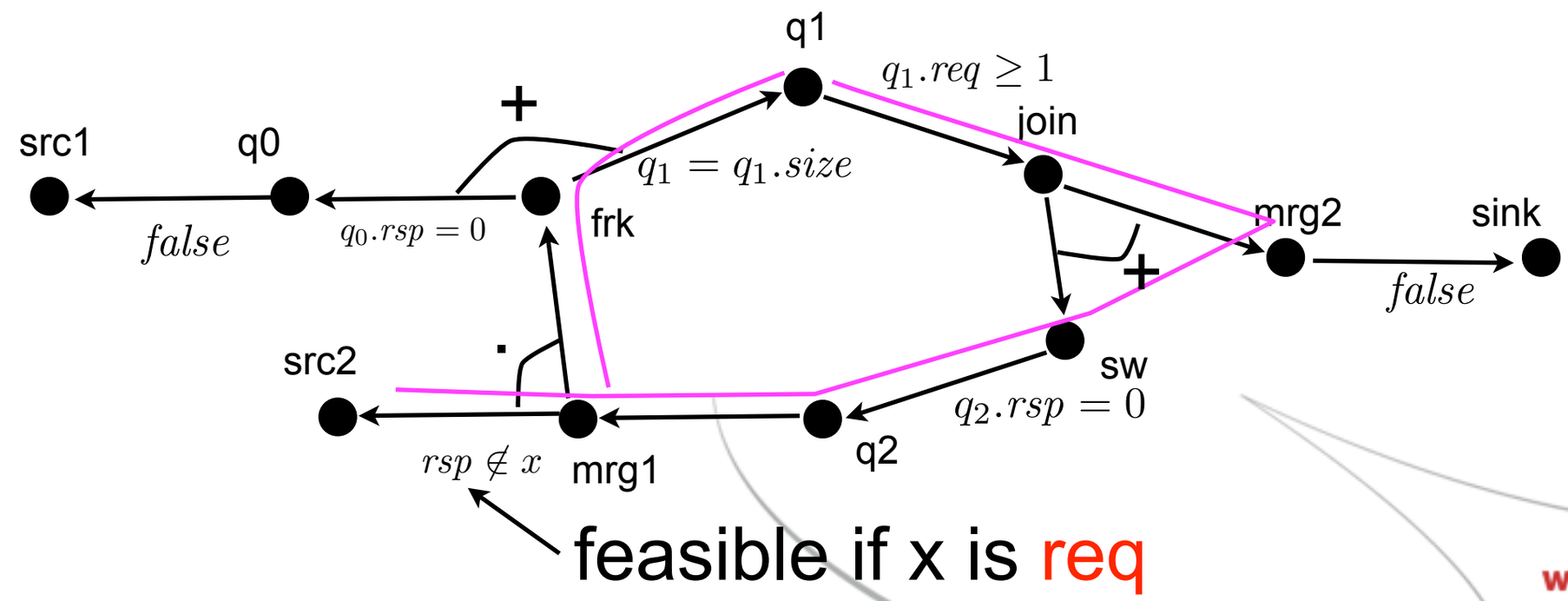




www.ou.nl

Step 2 / logically closed subgraph 2



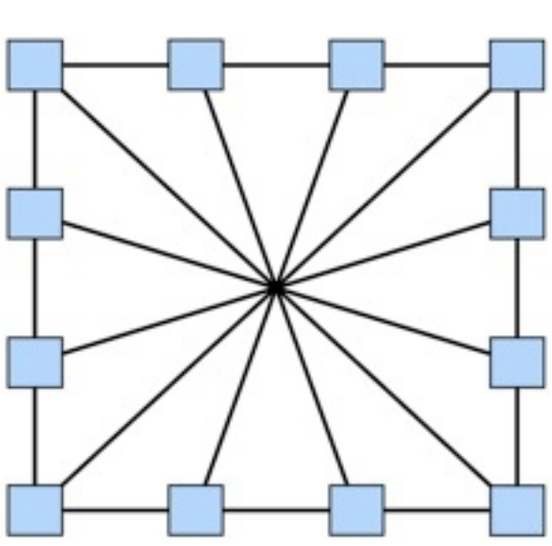


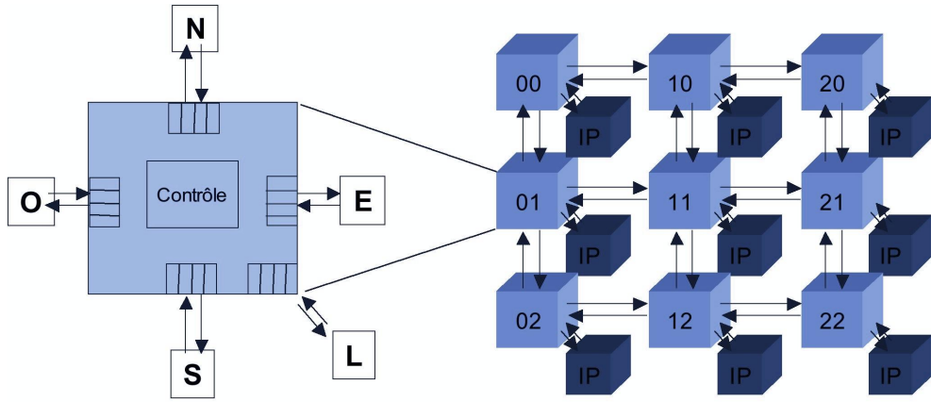
feasible if x is req

Implementation and case studies

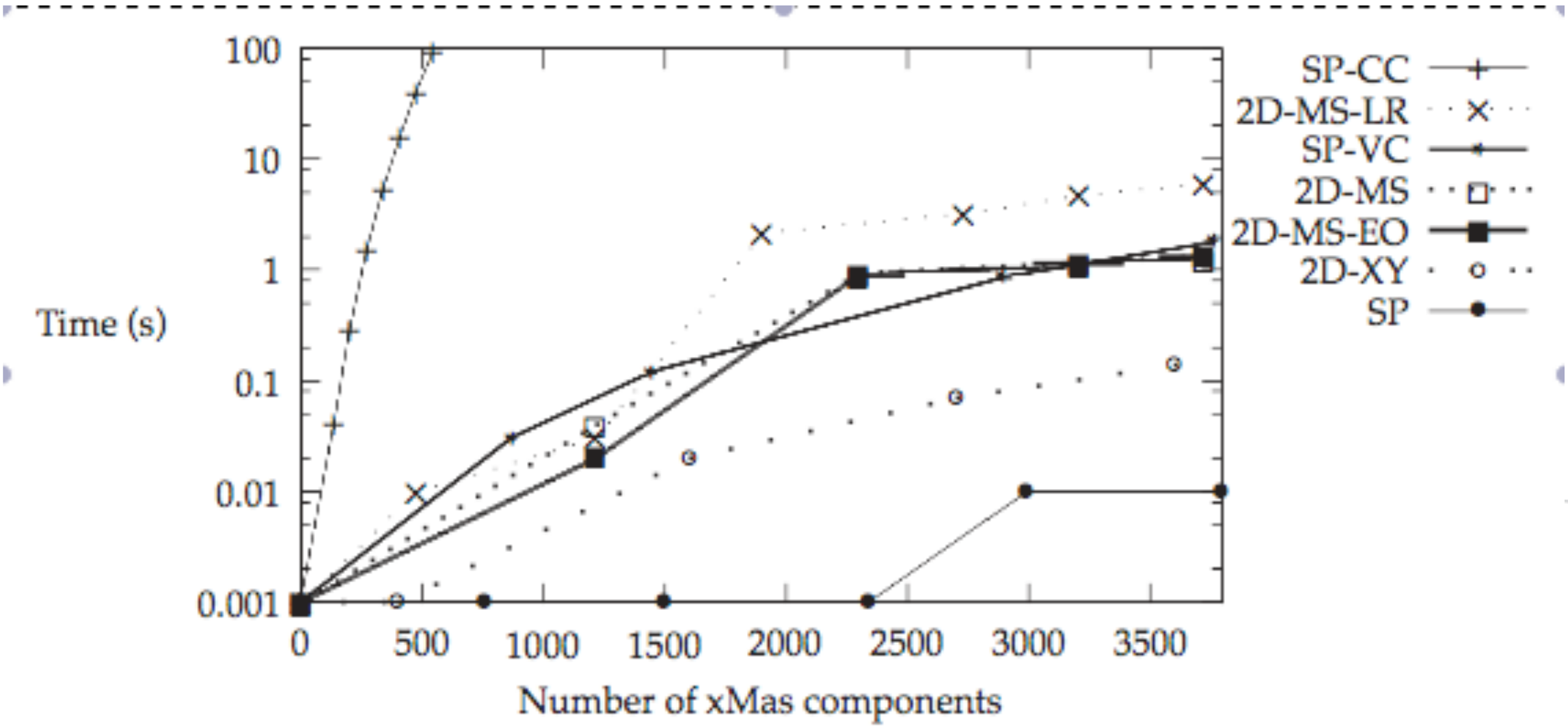
- Implementation of algorithm in C

- 2 topologies
  - Spidergon from STMicroelectronics
  - HERMES from Univ. Rio Grande (Brazil)





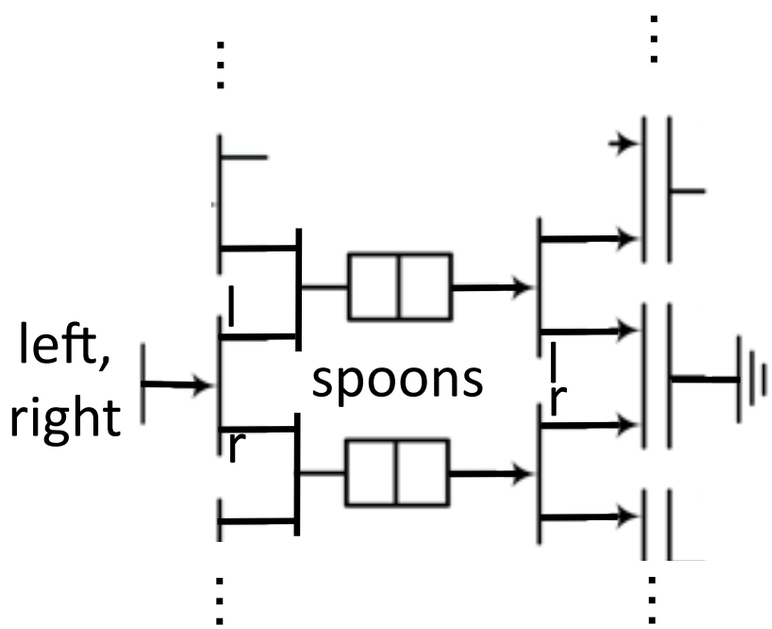
Experimental results



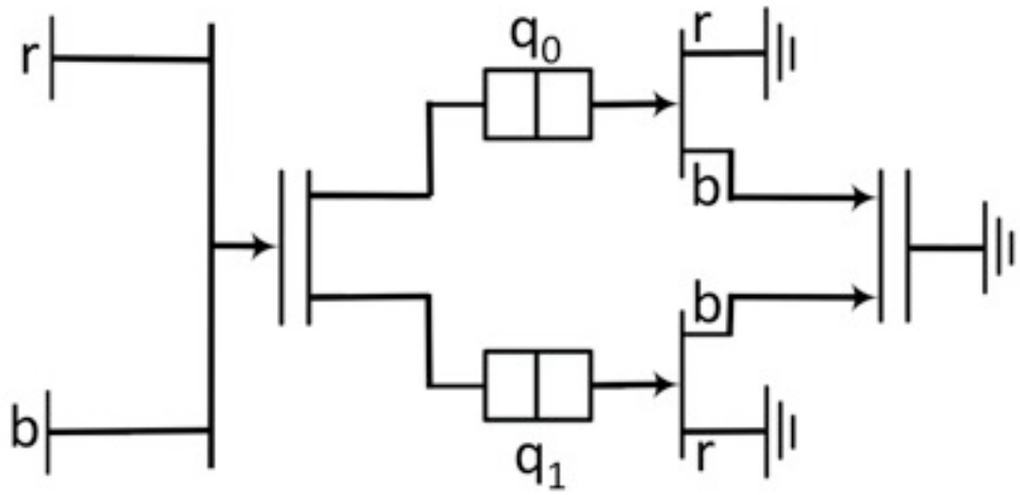
An academic example - Dining Philosophers

- Philosophers model in xMAS
  - Hands as 2 message types
  - Spoons as queues of size 1
  - "Eat" as join between hands

- A ring of philosophers
  - Easy problem for our algorithm
  - 3 000 philos in 0.06s



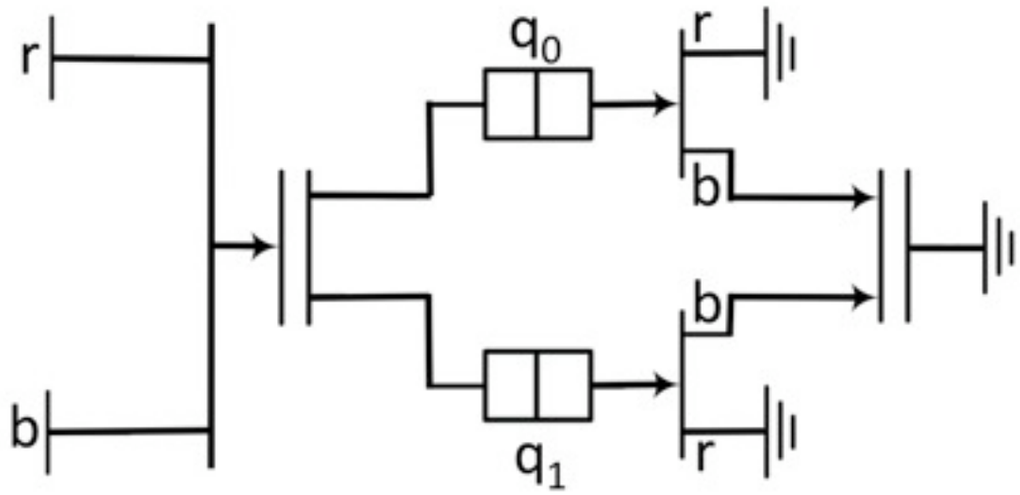
An example too hard for Intel but not for us !



- Two message types
  - blue and red
- Sorting queues
- Reds and blues synchronized before sink
- Is this network deadlock-free ?

An example too hard for Intel but not for us !

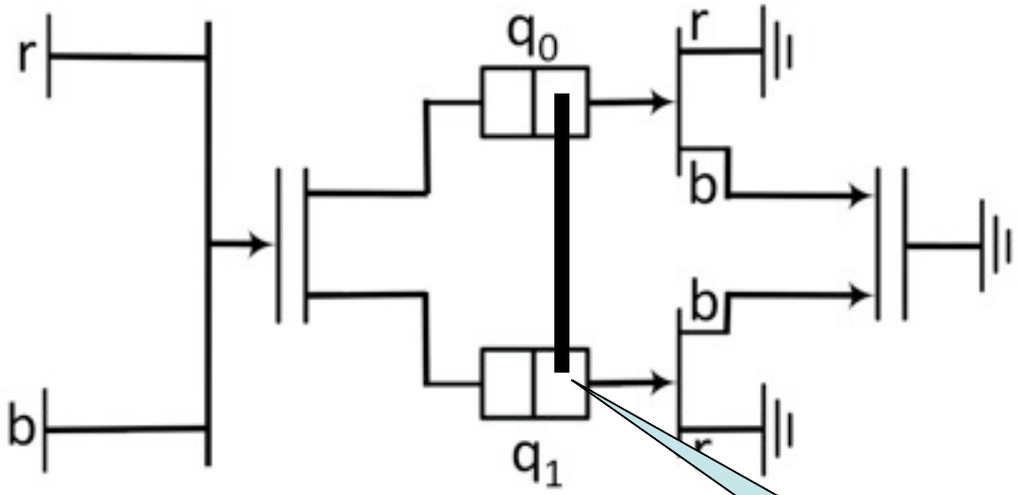
deadlock-free



- Two message types
  - blue and red
- Sorting queues
- Reds and blues synchronized before sink
- Is this network deadlock-free ?

An example too hard for Intel but not for us !

deadlock-free



ordering for blues and reds

- Two message types
  - blue and red
- Sorting queues
- Reds and blues synchronized before sink
- Is this network deadlock-free ?

Outline

- Intel's micro-architectural description language
  - xMAS definition
  - examples
- Deadlock verification for xMAS
  - definition of deadlocks
  - labelled dependency graph
  - feasible logically closed subgraph
- Conclusion and future work

Conclusion and future work

- Tool to detect message dependent deadlocks
  - Very efficient
  - Intel's our only concurrent
  - We can already handle more cases than Intel's techniques

- Still need to be formally proven
  - Connection with our previous work on GeNoC

- Composition/Hierarchy
  - Check sub-networks first and then compose

- Memory consistency proofs
  - e.g. Producer Consumer relations
  - Open PhD position sponsored by Intel/Open Universiteit/Radboud

Thanks !

Deadlock example 3

- Channels with three signals
  - data, input ready, target ready
- Transfer cycle
  - both input and target are "true"