Hunting Deadlocks Efficiently in Micro-Architectural Models of Communication Fabrics

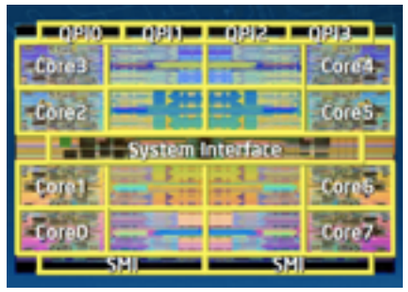
Freek Verbeek and Julien Schmaltz

Radboud University Nijmegen

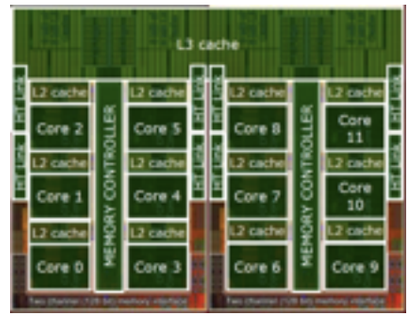
Open Universiteit
www.ou.nl

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

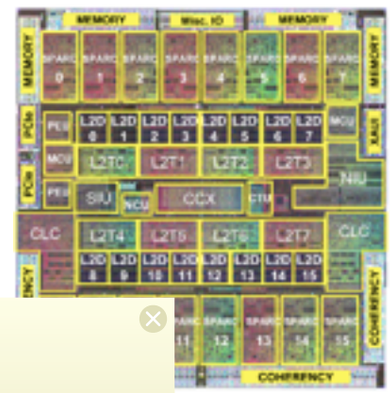
Intel 8 cores
~2.3 Bill. T. on 6.8cm^2



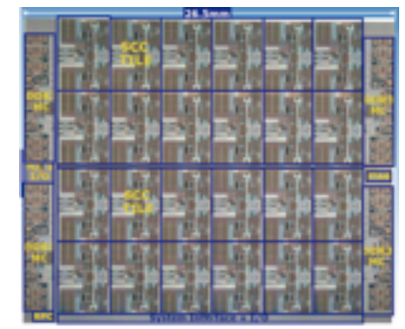
AMD Opteron 12 cores
~1.8 Bill. T. on 2x3.46cm^2



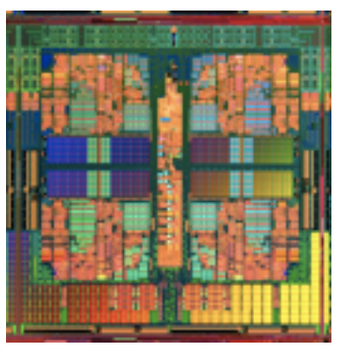
Sun Niagara3 16 cores
~1 Bill. T. on 3.7cm^2



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



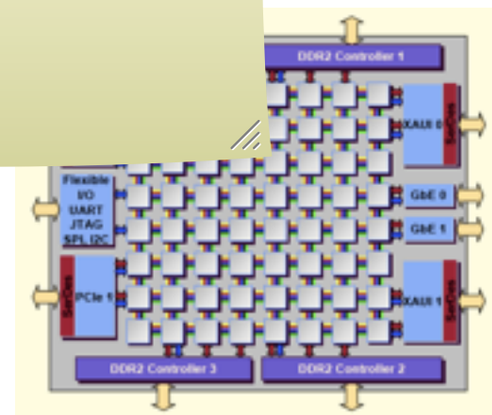
Intel 4 cores
~582 Mio. T. on 2.86cm^2



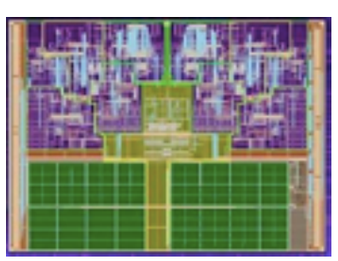
Intel Research 80 c
~100 Mio. T. on 2.7

Usual:
- verify cores
- verify interconnect

ro64 64 cores

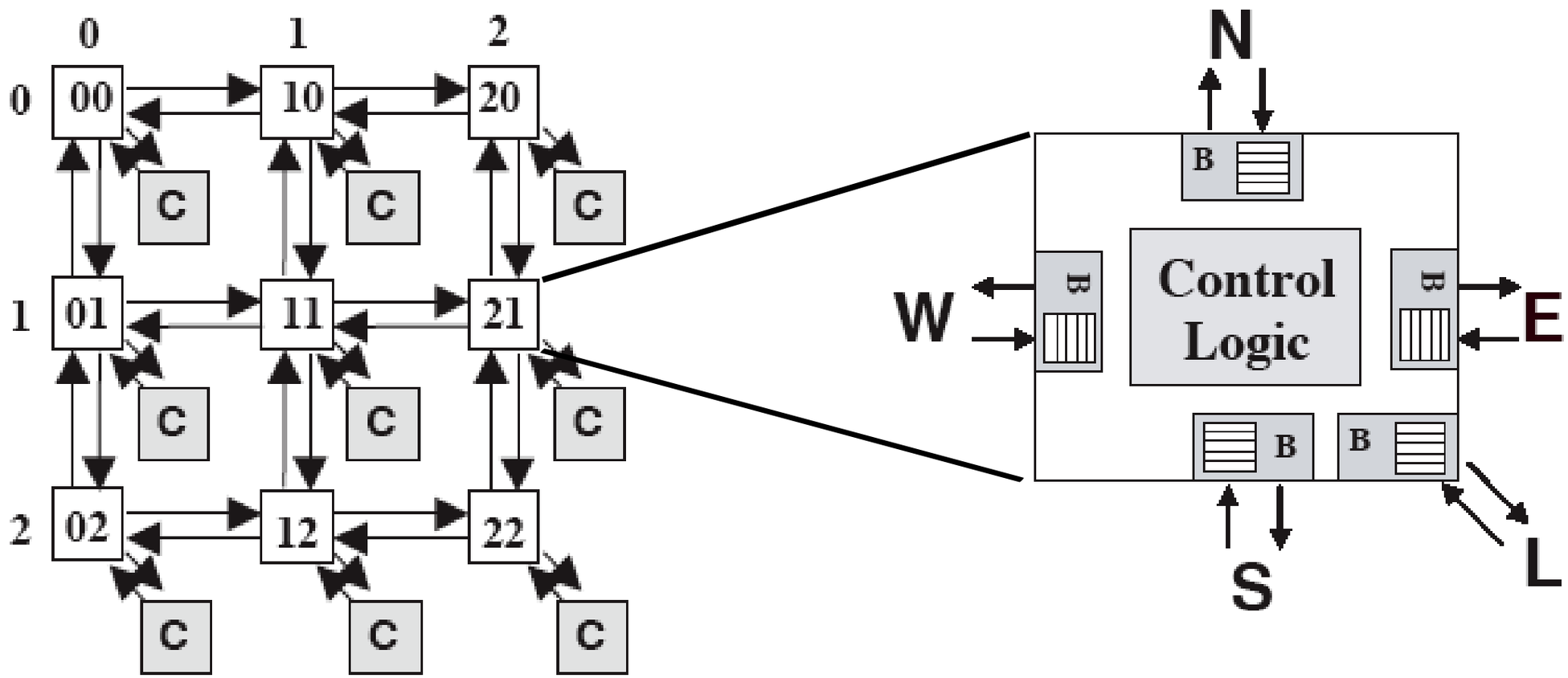


Intel 2 cores
~167 Mio. T. on 1.1cm^2



Networks-on-Chips: Example 1, HERMES

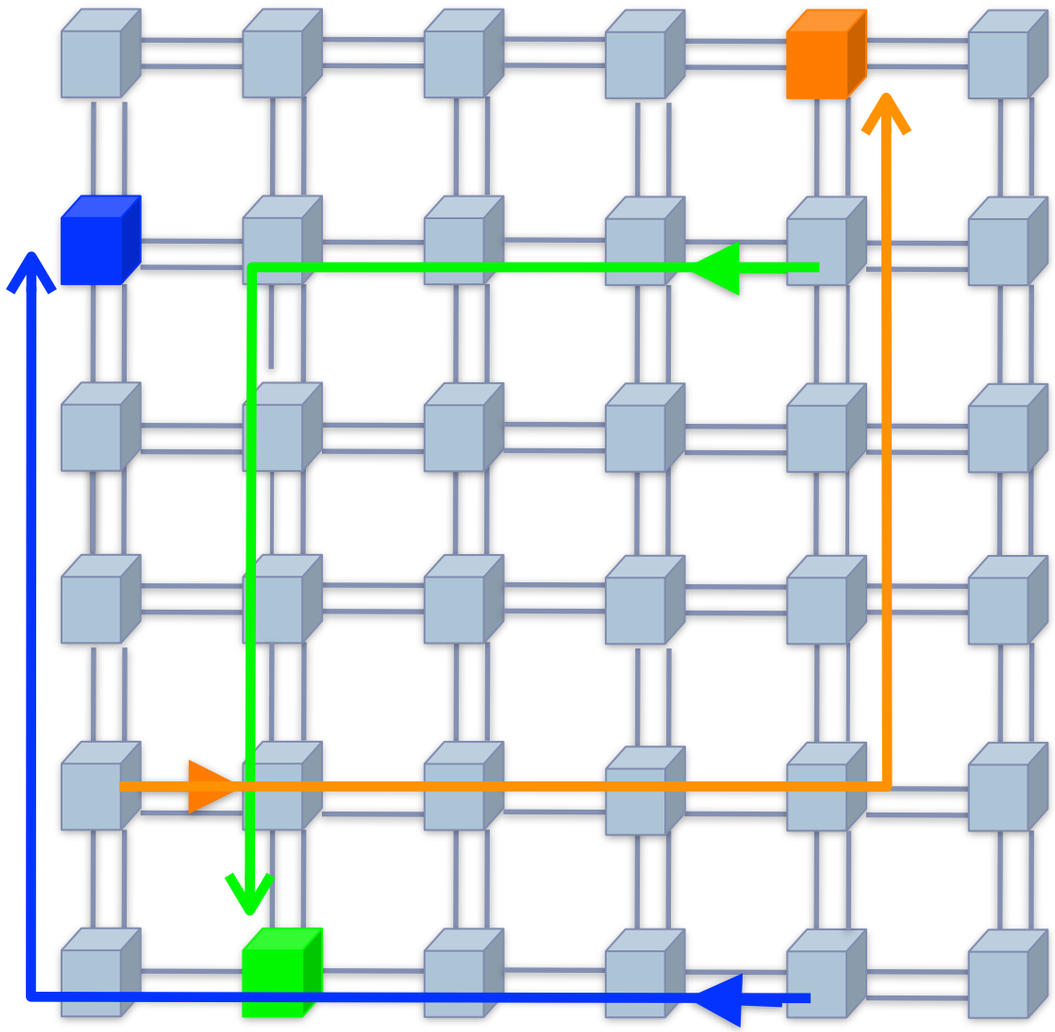
The topology:



- Two dimensional mesh

Networks-on-Chips: Example 1

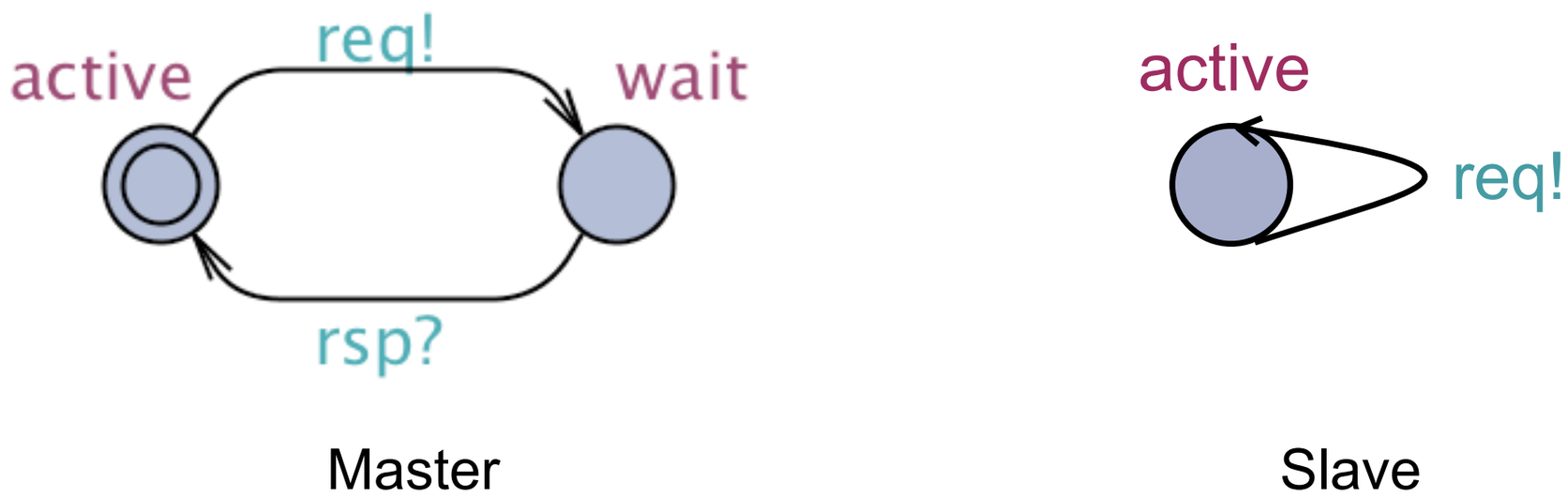
The routing function:



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

Networks-on-Chips: Example 1

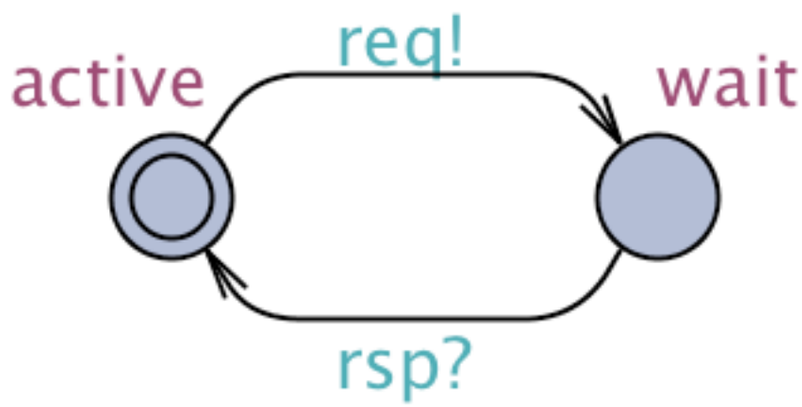
The high-level protocol:

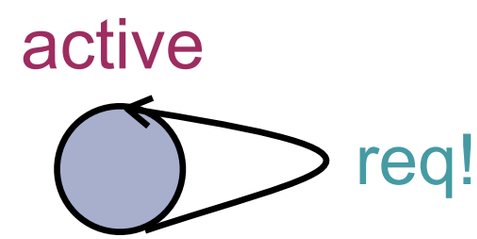


- Masters send requests and wait for responses
- Slaves produce responses when receiving requests
- Deadlock-free protocol

Networks-on-Chips: Example 1

The high-level protocol:





Master

Slave

- No message dependencies

$$rsp \prec req \wedge req \not\prec rsp$$

Networks-on-Chips: Example 1

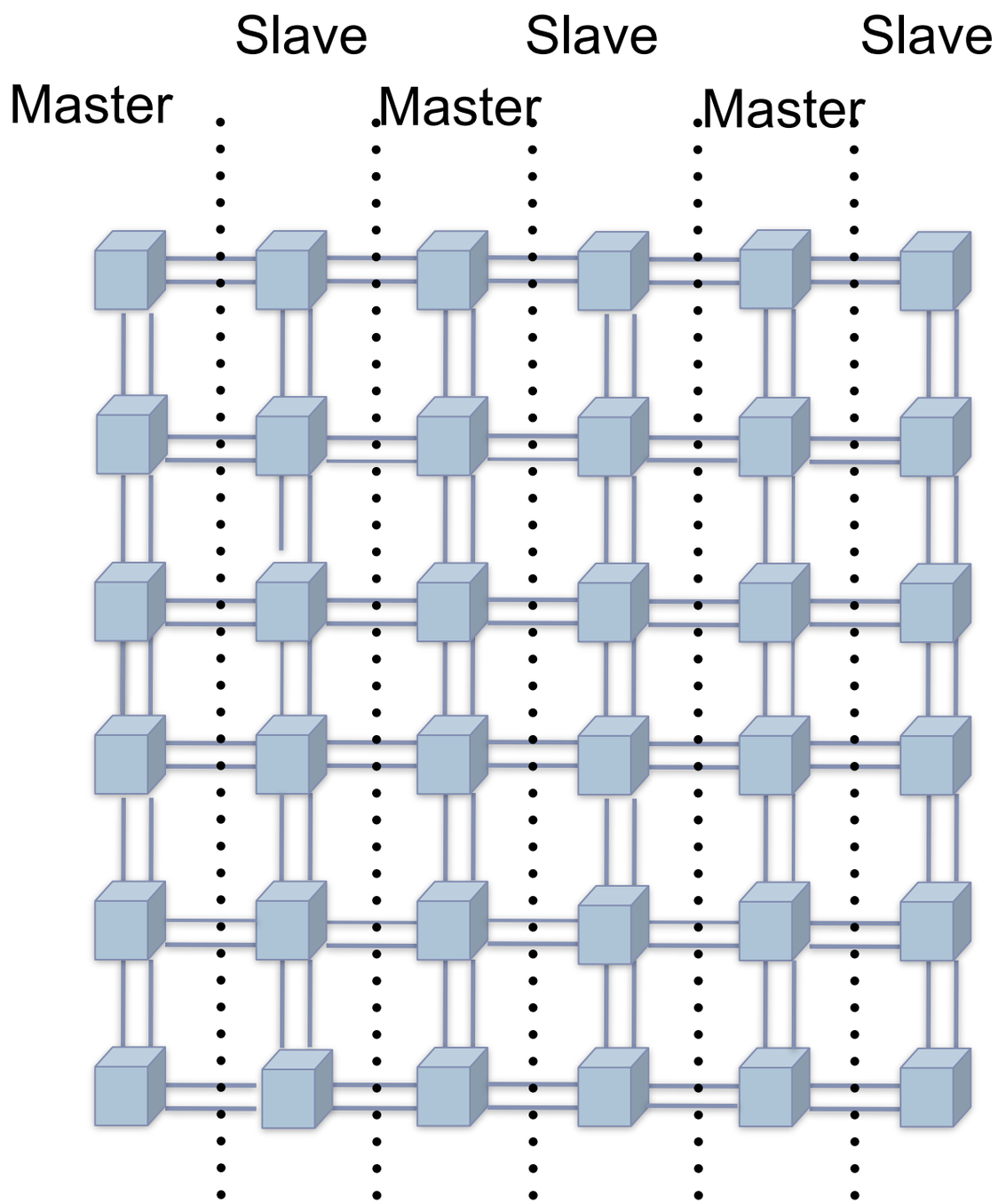
Network component	Deadlock-free?
Topology	✓
Routing Function	✓
High-level protocol	✓
Message Dependencies	✓

$\stackrel{?}{=}$

Deadlockfree system

Networks-on-Chips: Example 1

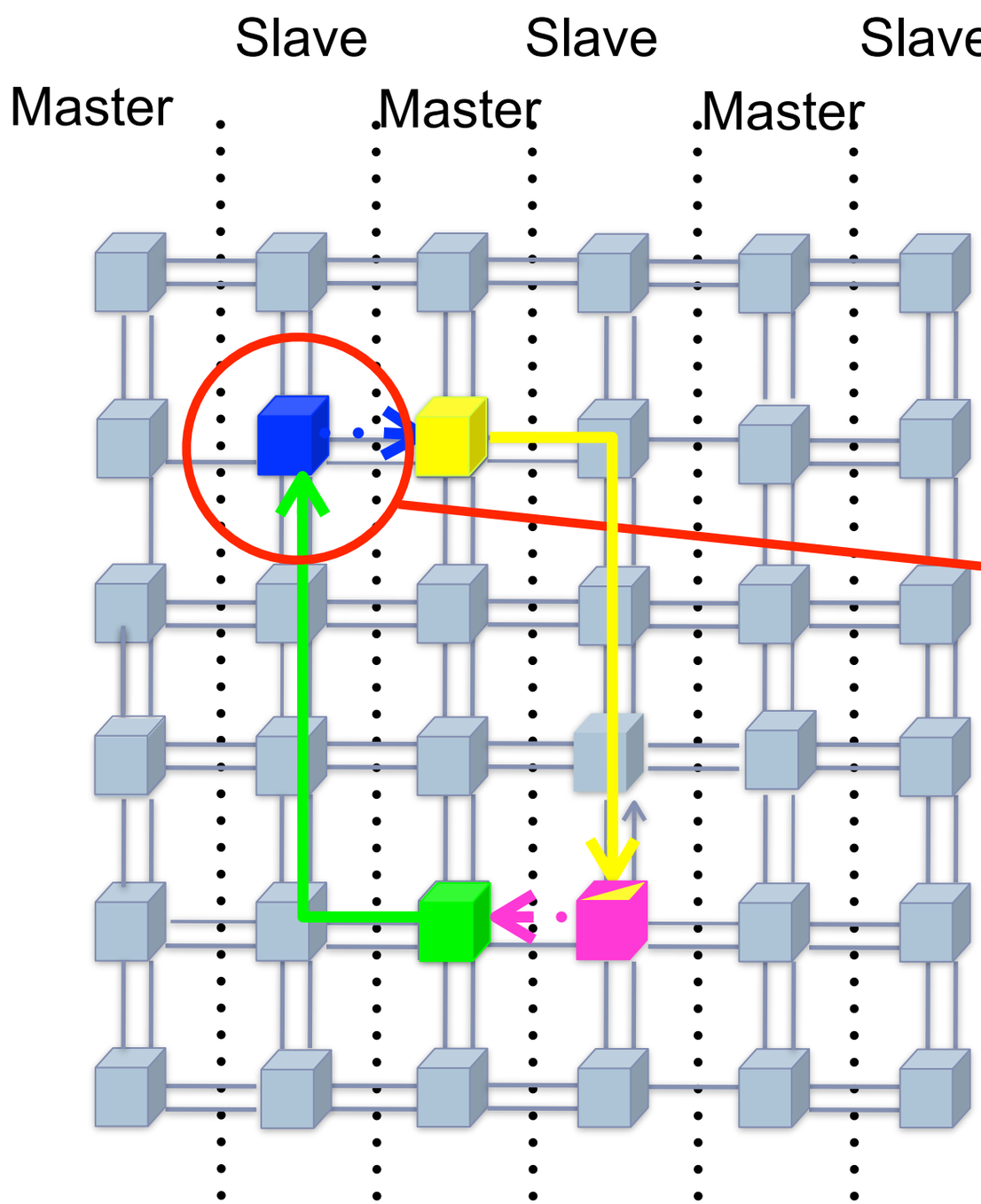
Core distribution:

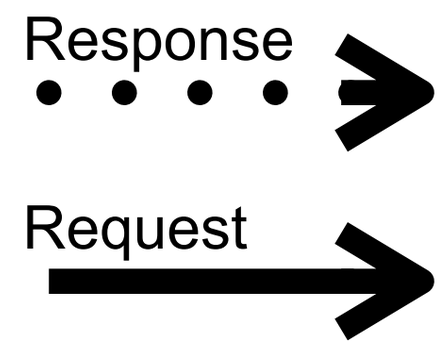


- Masters on the odd/slaves on the even columns

Networks-on-Chips: Example 1

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





Green requests waits for blue reponses

Networks-on-Chips: Example 1

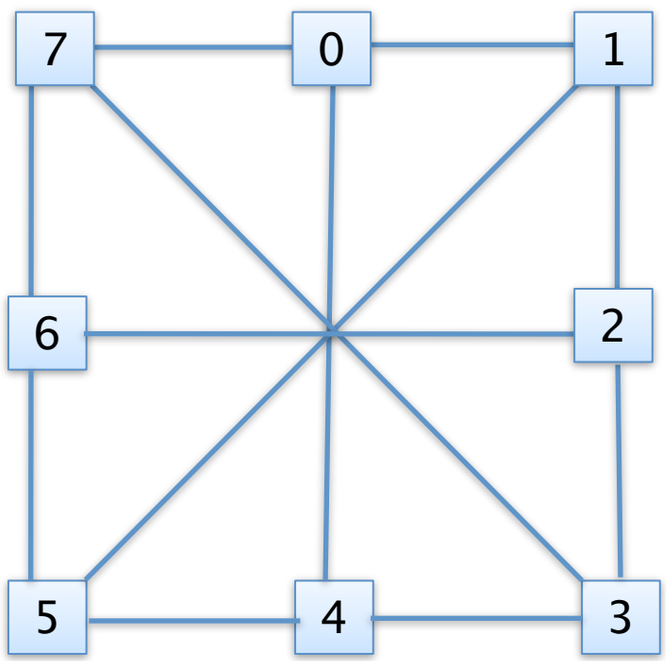
Network component	Cause of deadlock?
Topology	✓
Routing Function	✓
High-level protocol	✓
Message Dependencies	✓

=

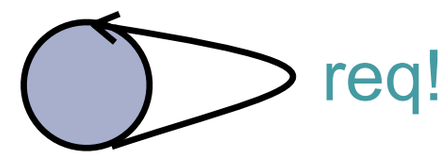
Deadlockfree system ✗

Networks-on-Chips: Example 2, Spidergon from STMElectronics

Topology



High-level protocol



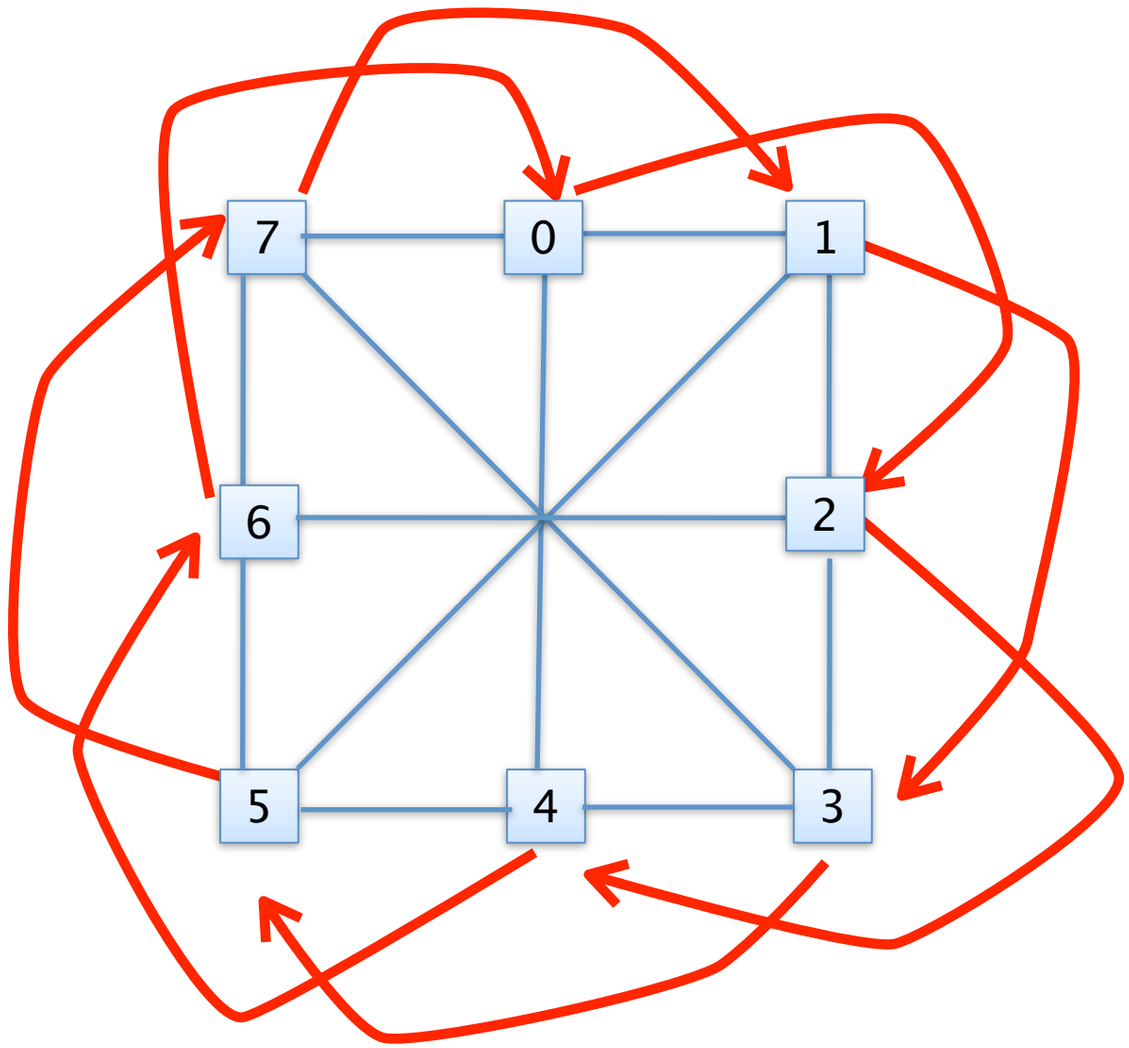
Routing logic

```
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
```

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

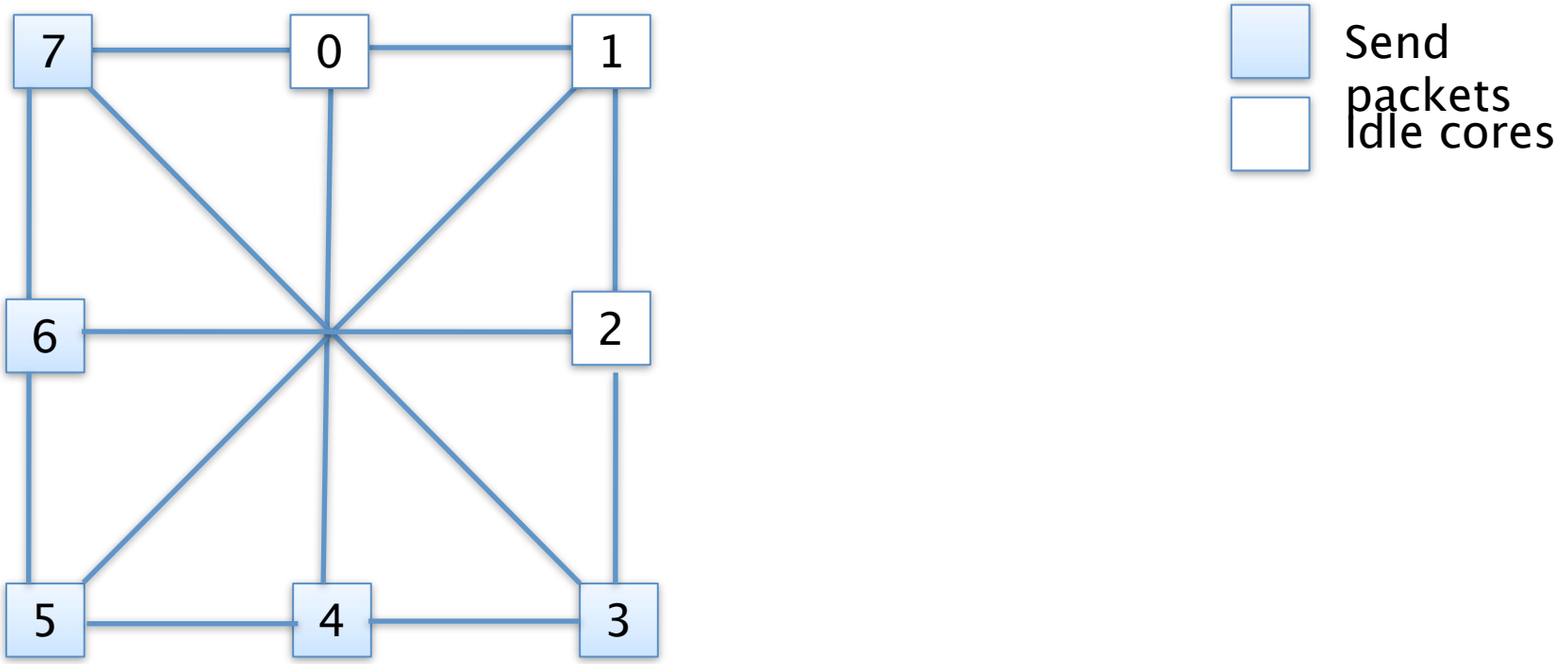
Networks-on-Chips: Example 2

Network component	Cause of deadlock
Routing Function	✗



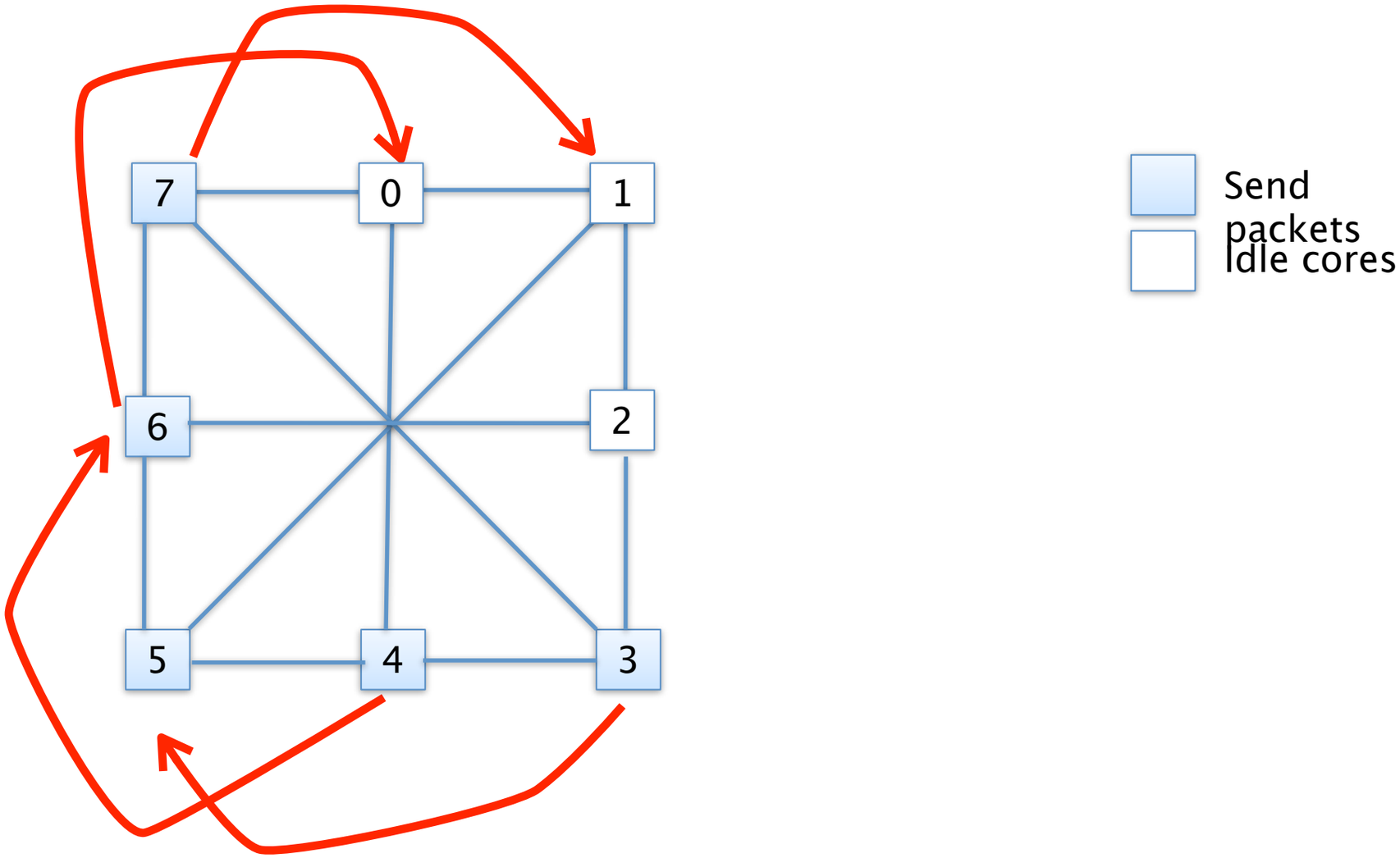
Networks-on-Chips: Example 2

- Is the system deadlock-free ?



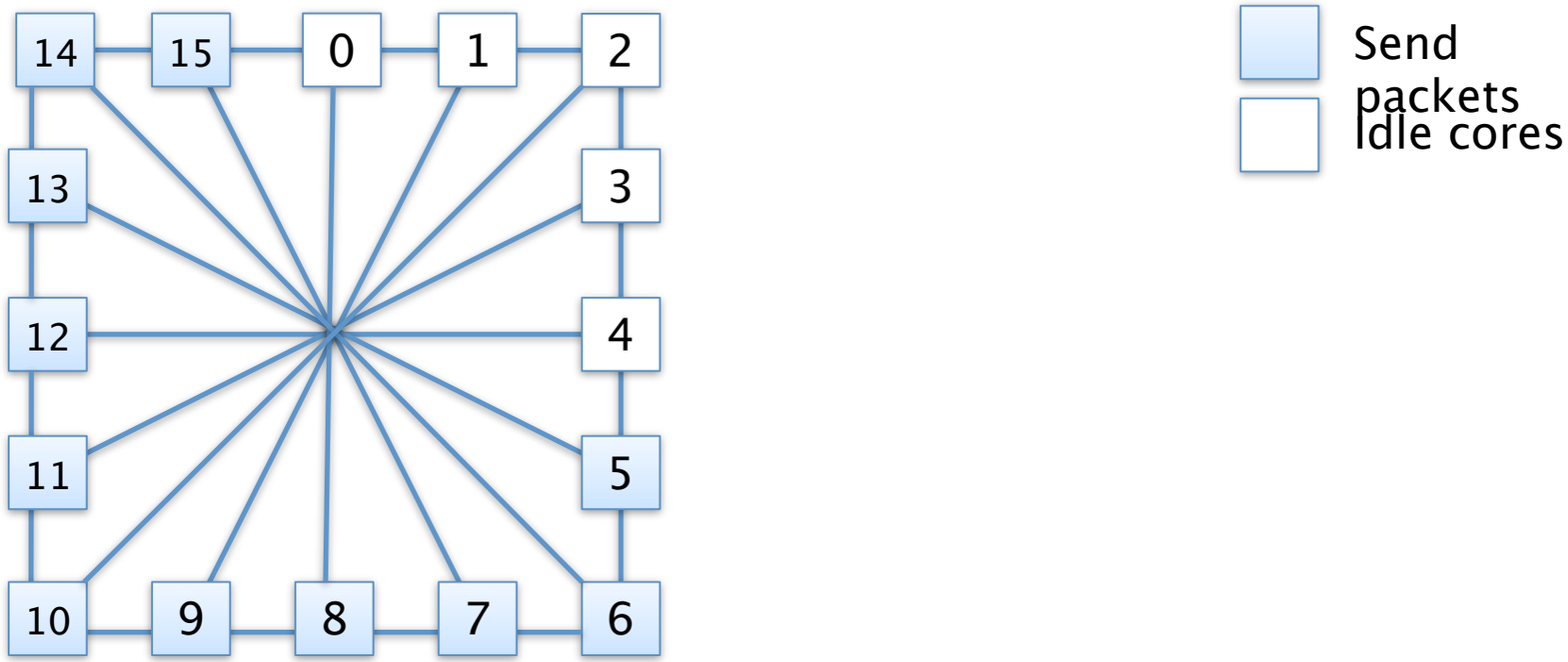
Networks-on-Chips: Example 2

- Is the system deadlock-free ?
- Yes ! None of the dependencies in the right upper quarter occur.



Networks-on-Chips: Example 2

- Is the system deadlock-free ?



Networks-on-Chips: Example 2

Network component	Deadlock-free?
Topology	✓
Routing Function	✗
High-level protocol	✓
Message Dependencies	✓
Core Distribution	✓ ✗
Network size	✓ ✗

=

Deadlockfree system ✓ ✗

Networks-on-Chips: Example 3

Network component	Deadlock-free?
Topology	✓ ✗
Routing Function	✓ ✗
High-level protocol	✓ ✗
Message Dependencies	✓ ✗
Core Distribution	✓ ✗
Network size	✓ ✗
Queue sizes	✓ ✗
Counter information	✓ ✗
Virtual channel allocation	✓ ✗

$\stackrel{?}{=}$

Deadlockfree system

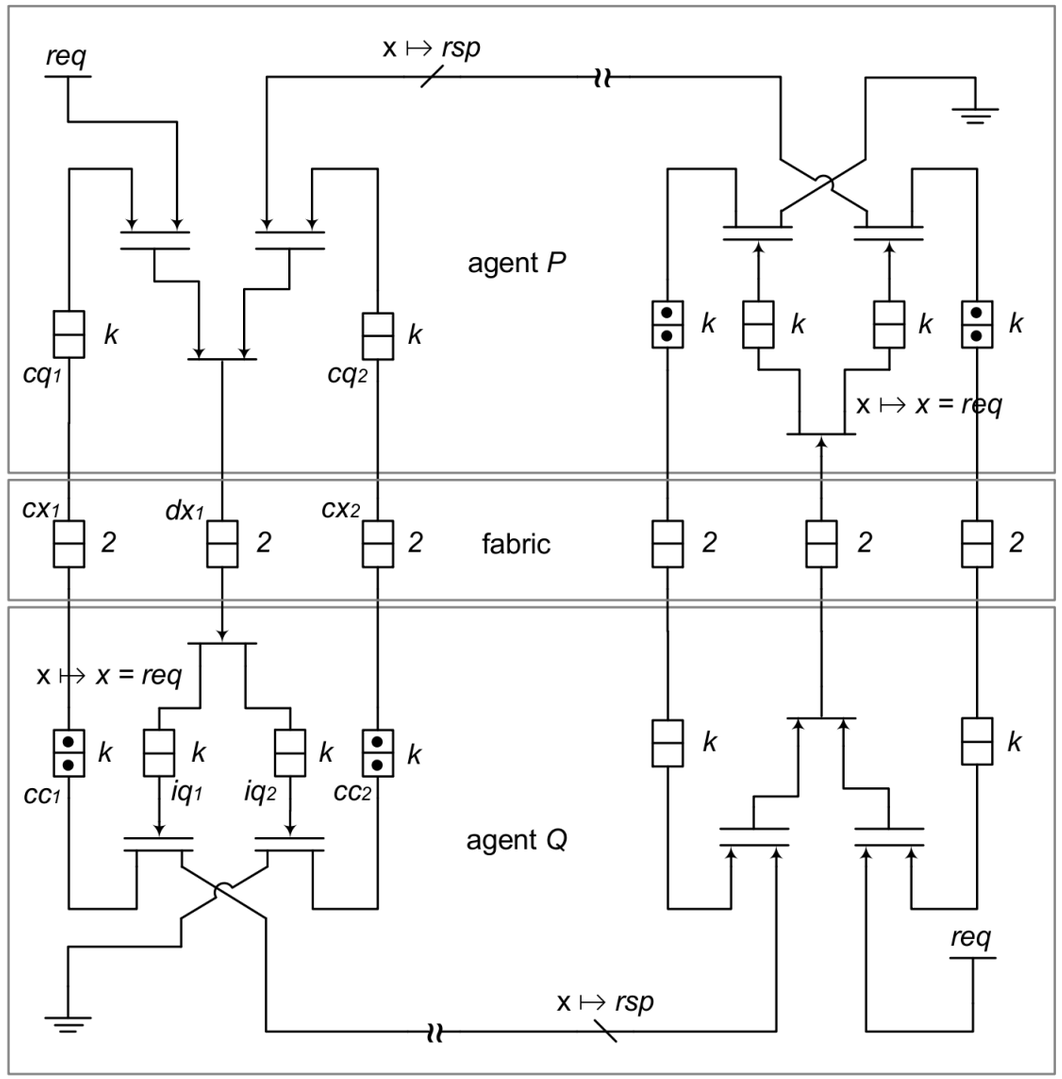
Confusing ...

- We need tools to (quickly) check for deadlocks
  - in large systems
  - with message dependencies
  - with the topology, routing and core behavior in one model
  - able to handle parameters such as queue size

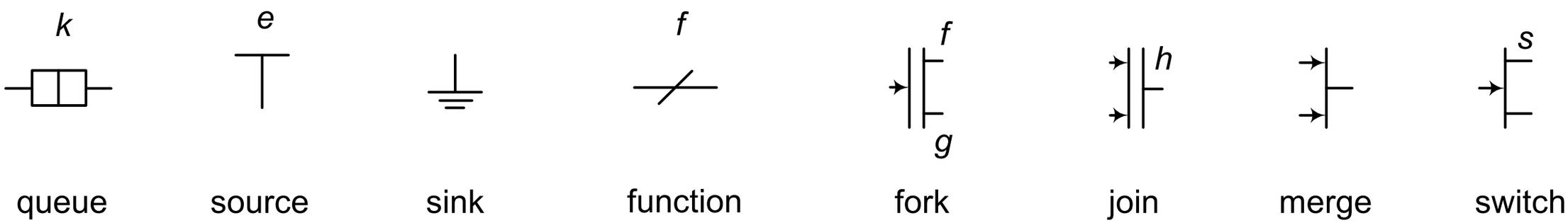
Outline

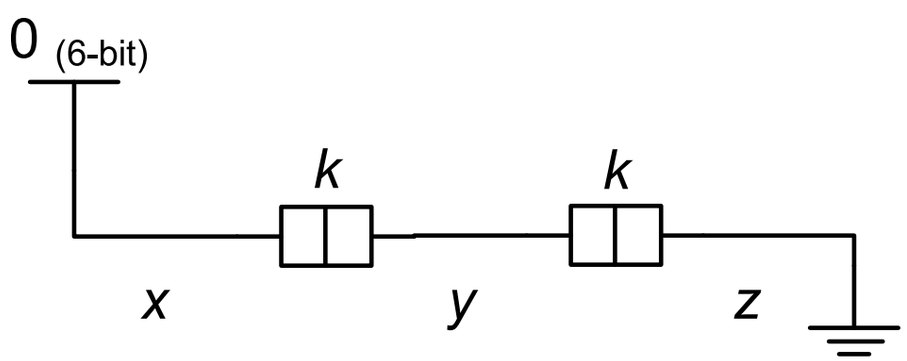
- Intel's micro-architectural description language
  - xMAS language
  - Capturing high-level structure and message dependencies
- Deadlock verification for xMAS
  - Definition of deadlocks
  - Labelled waiting graph
  - Feasible logically closed subgraph
- Conclusion and future work

Intel's abstraction for communication fabrics



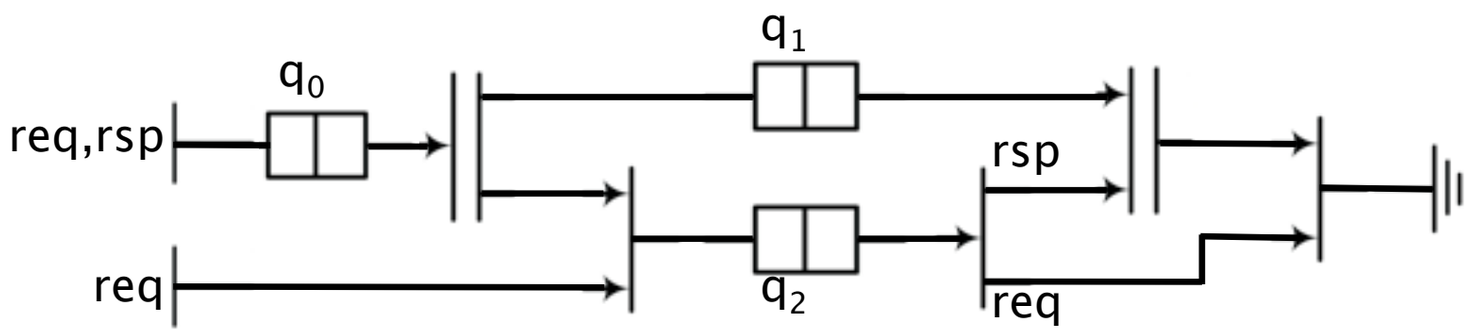
xMAS - Executable MicroArchitectural Specifications



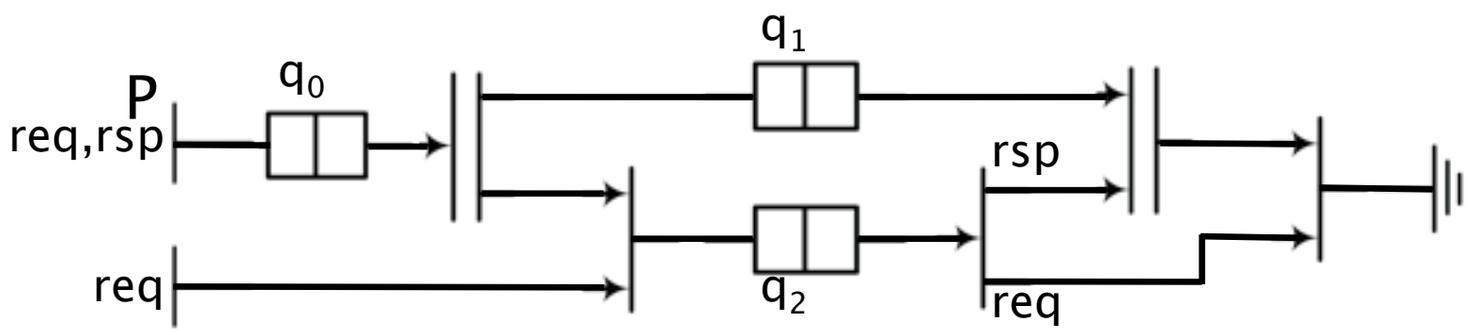


- Fair sinks and sometimes sources
- Diagram is formal model
- Friendly to microarchitects

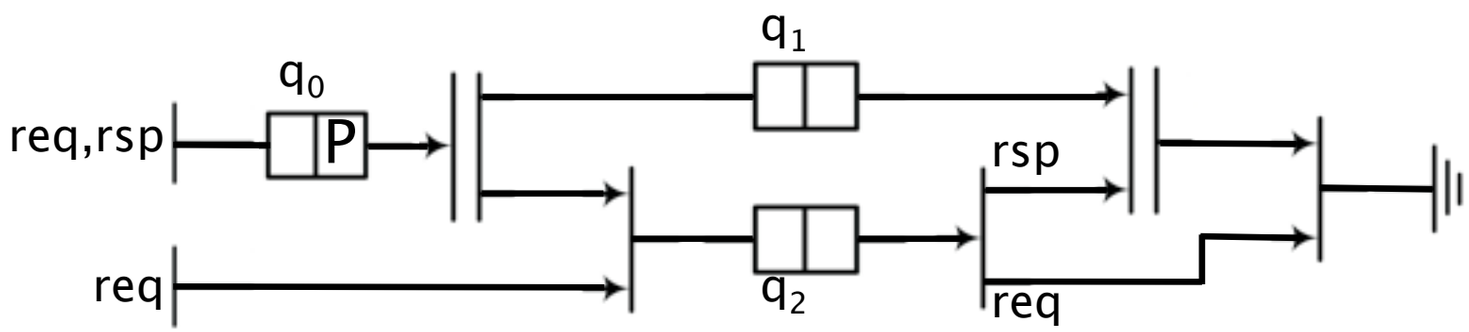
xMAS example



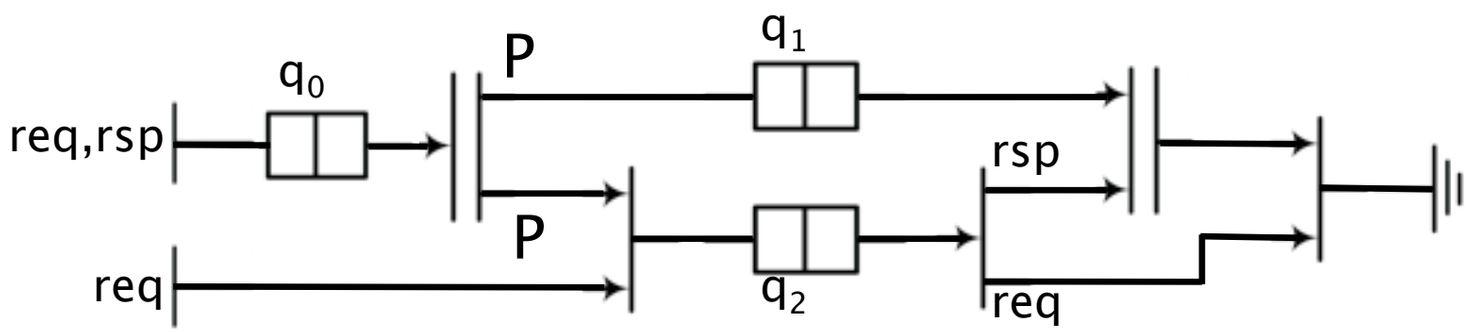
xMAS example



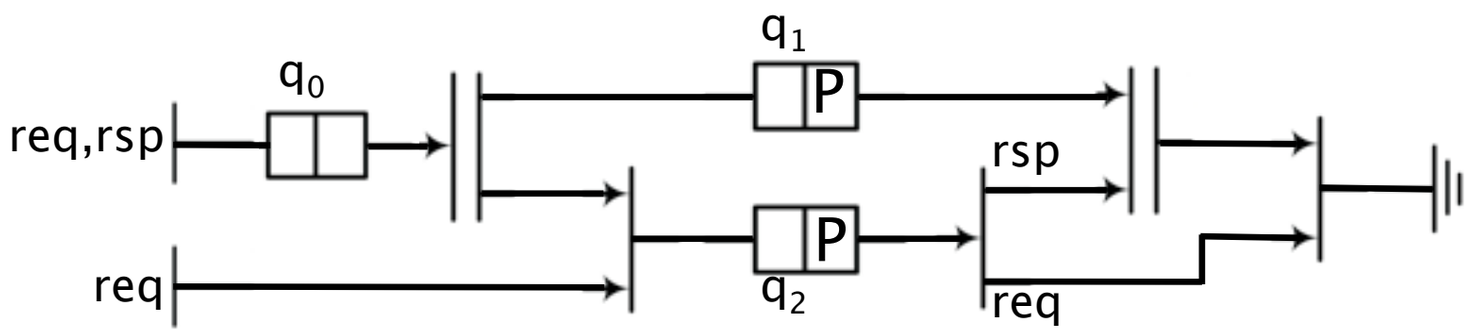
xMAS example



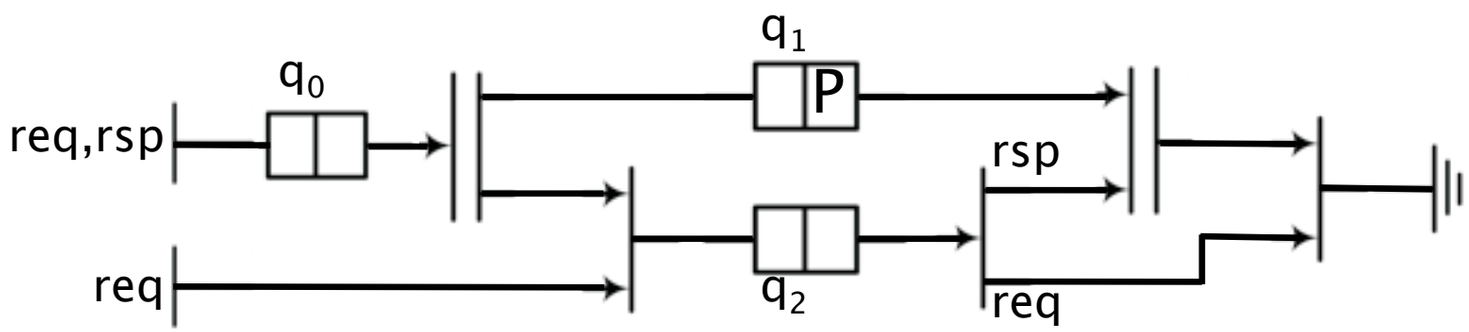
xMAS example



xMAS example



xMAS example



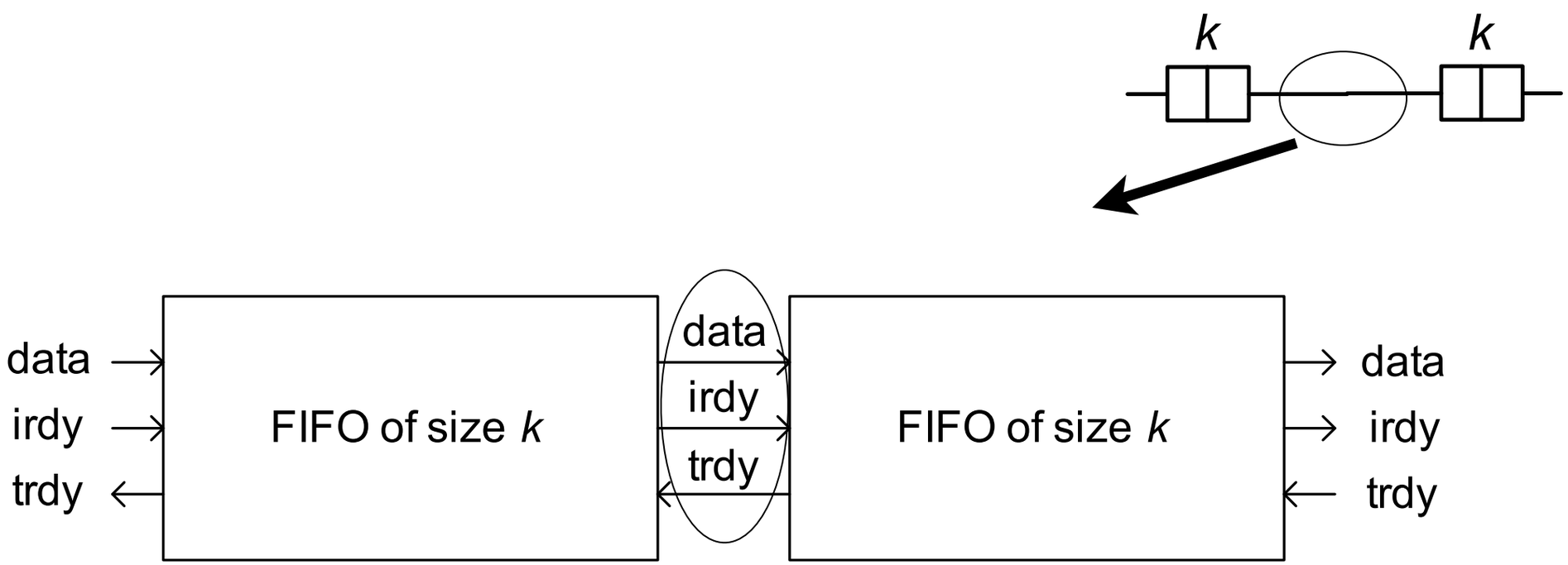
Outline

- 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

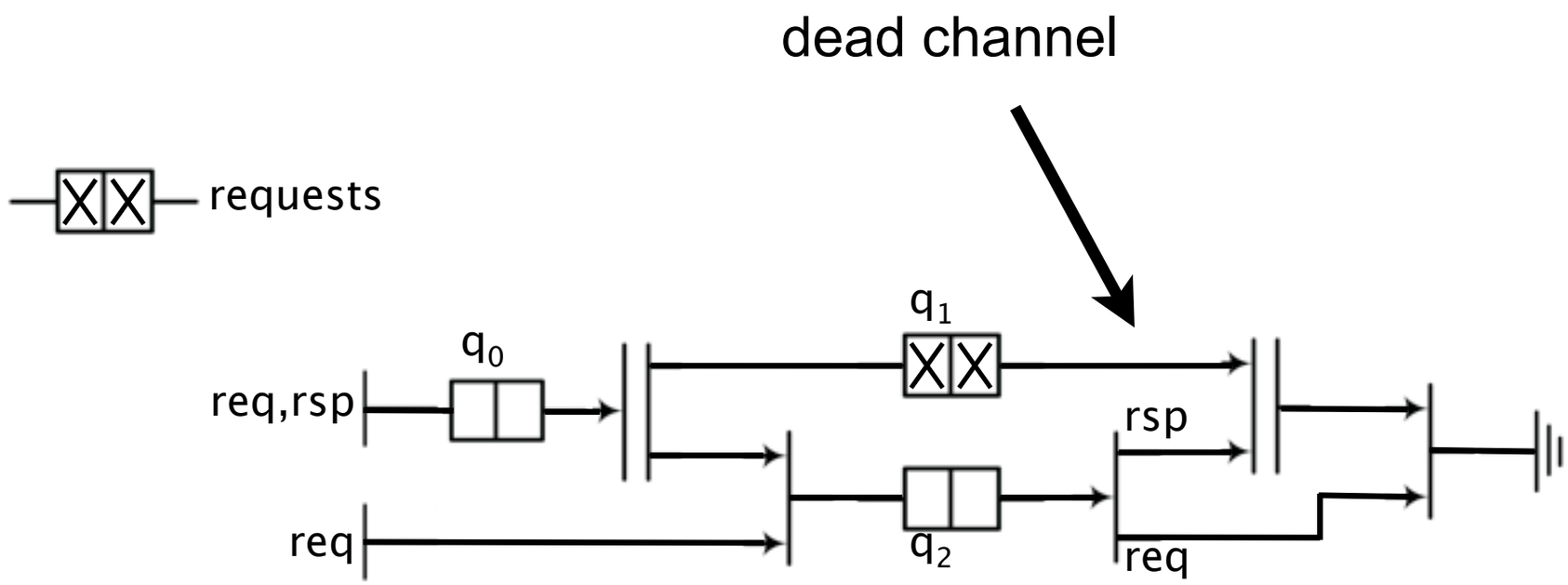
Formal definition of "deadlock" in xMAS

- Intuition is a "dead" channel
- Formal definition based on Linear Temporal Logic
  - Predicate logic
  - Temporal operators "eventually" ($\Diamond$) and "globally" ($\Box$)
- Channel *c* is dead iff

$$\Diamond(c.irdy \wedge \Box\neg c.trdy)$$



xMAS example



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

General approach for deadlock detection in xMAS networks

- Define Blocking 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

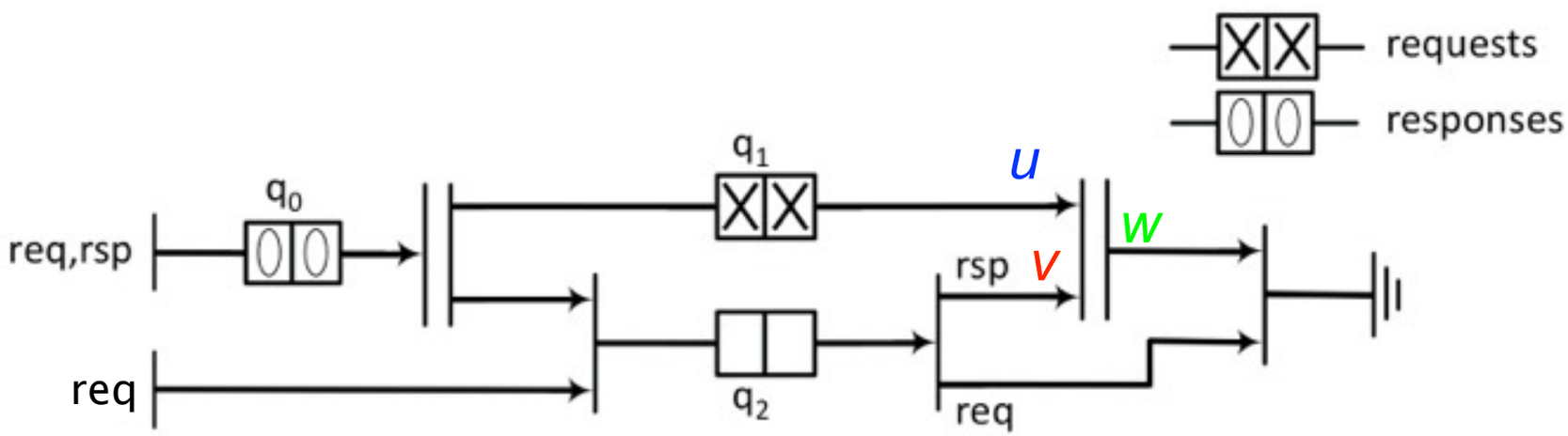
General approach for deadlock detection in xMAS networks

- Define Blocking 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

Blocking Equations for a join

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

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

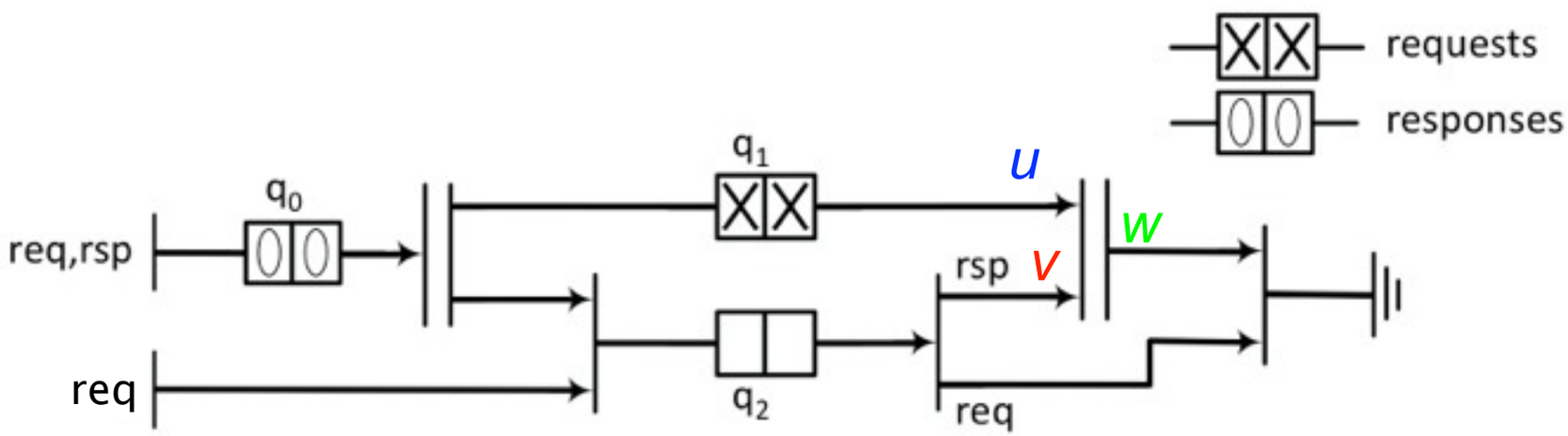


Blocking Equations for a join

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

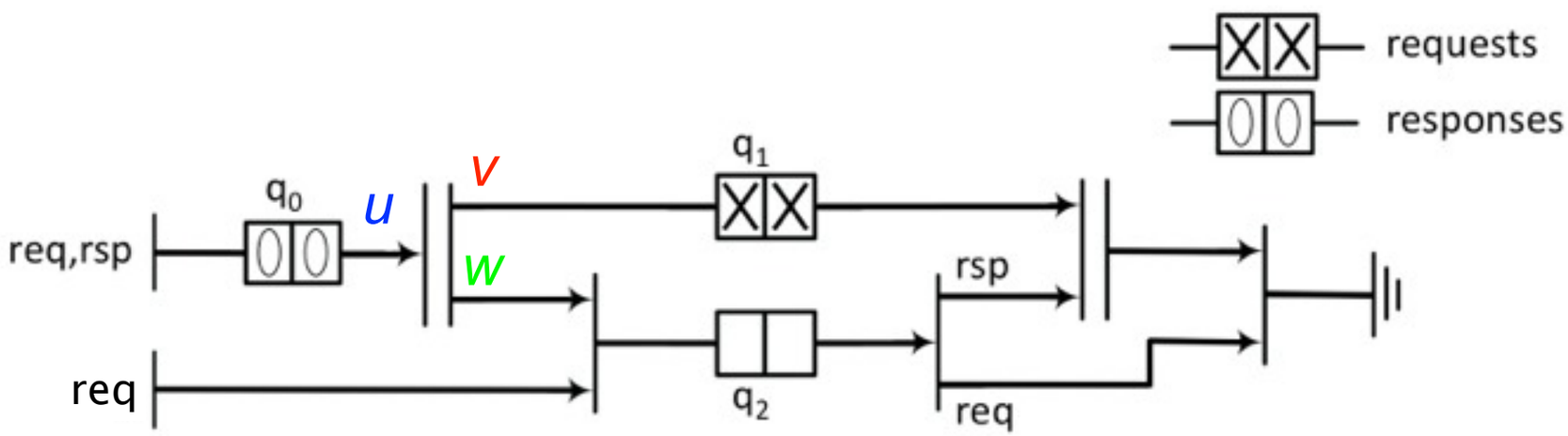
We need to know when a channel is idle !

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



Idle equations for a fork

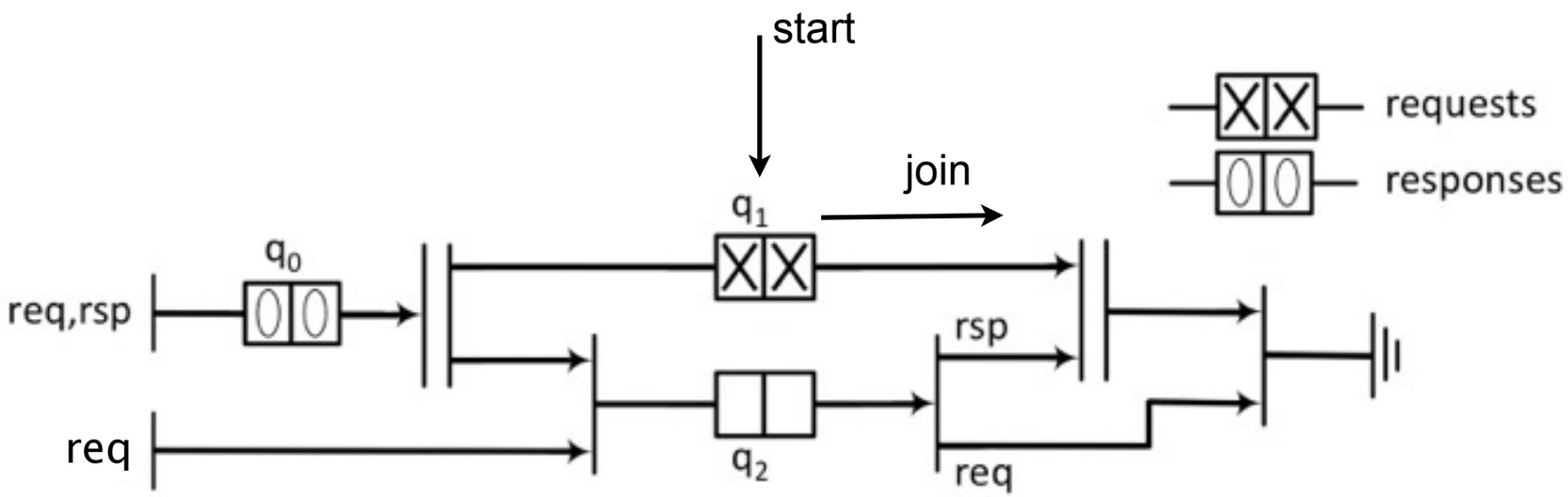
- A fork output is idle if the input is idle or the other output is blocked
- **Idle**(w) = **Idle**(u) + **Block**(v)

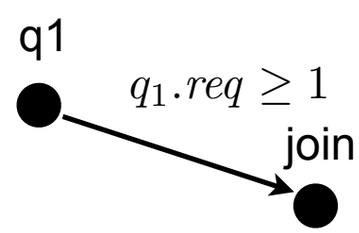


General approach for deadlock detection in xMAS networks

- Define Blocking 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

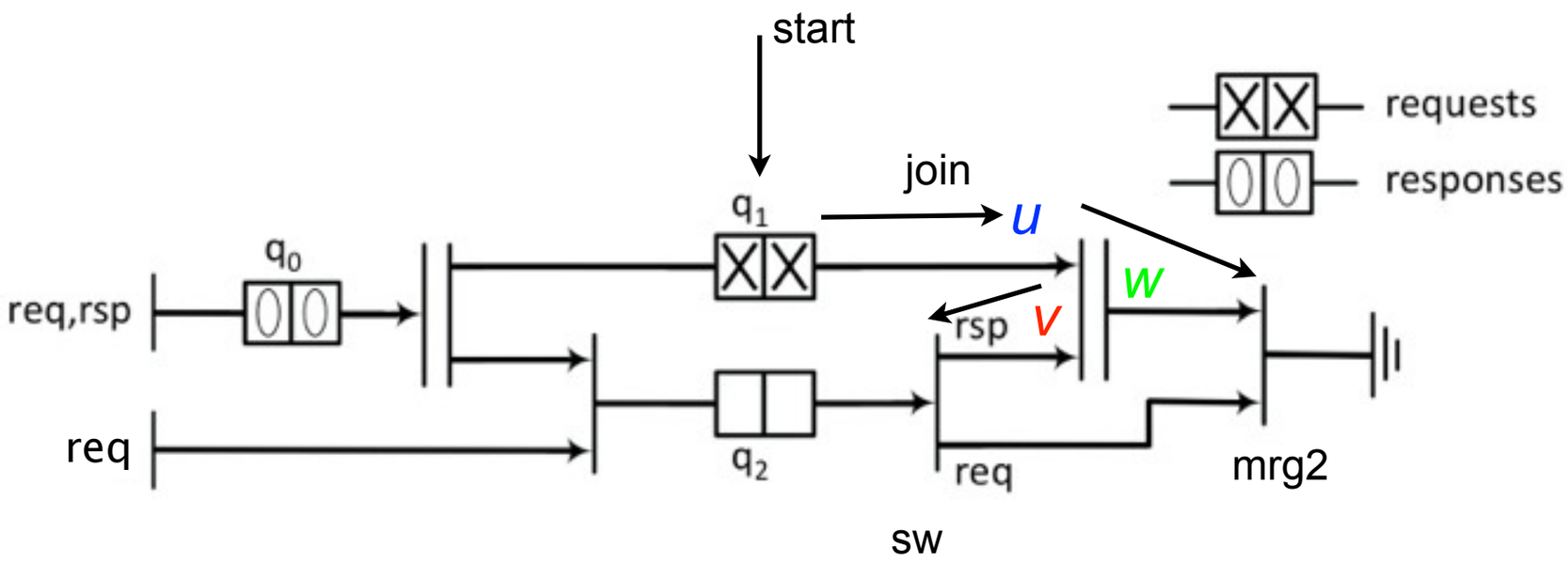
Step 2 / labelled dependency graph (1)



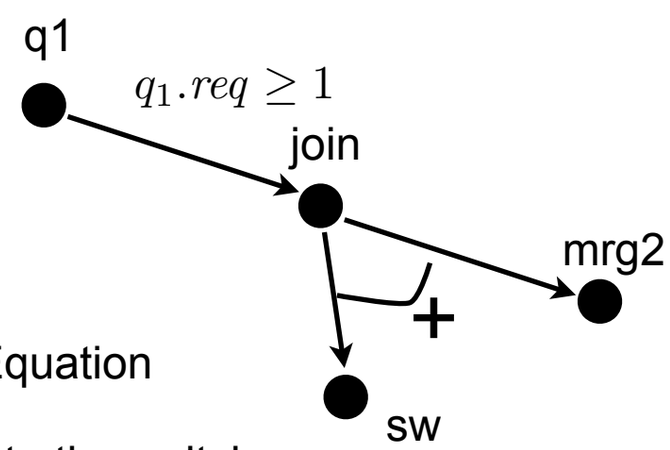


start with a message in q1 and visit the join

Step 2 / labelled dependency graph (2)



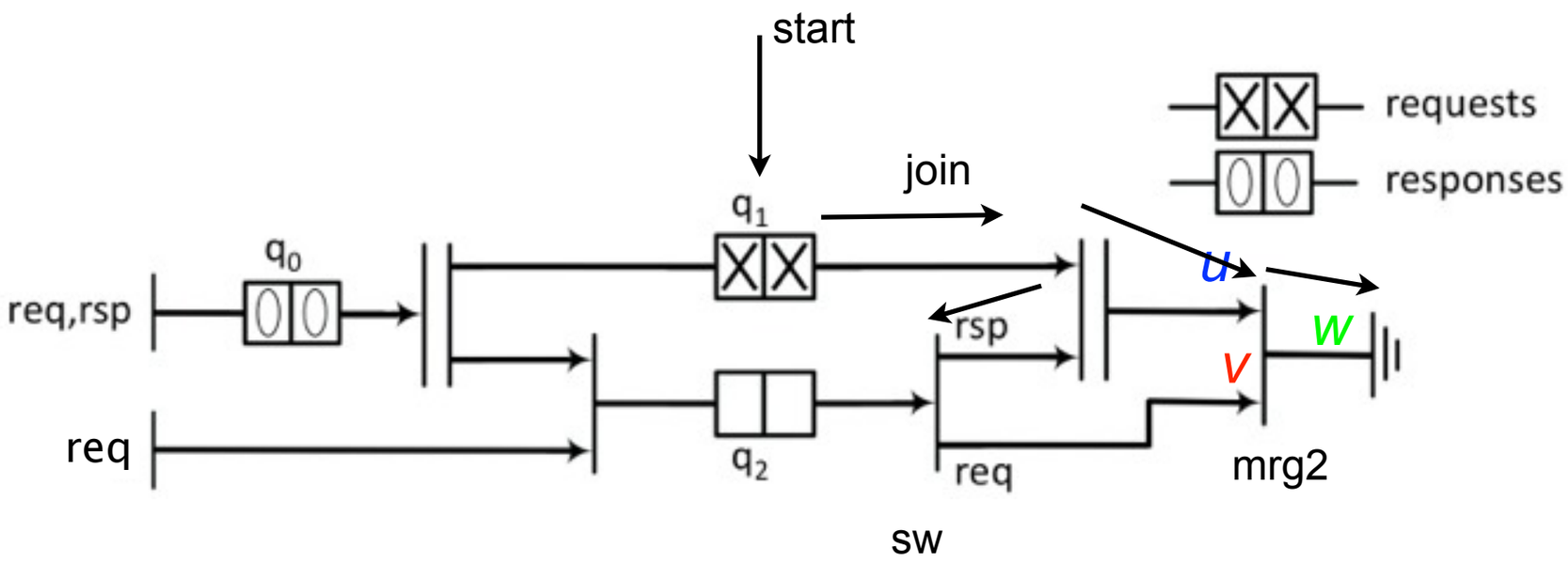
Block(u) = **Idle**(v) + **Block**(w)



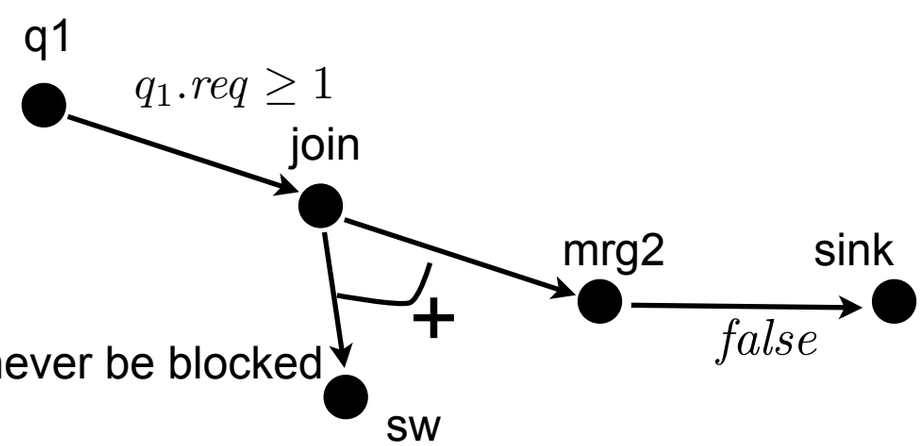
analyse the join according to its Blocking Equation

we go forward to the merge and backward to the switch

Step 2 / labelled dependency graph (2)



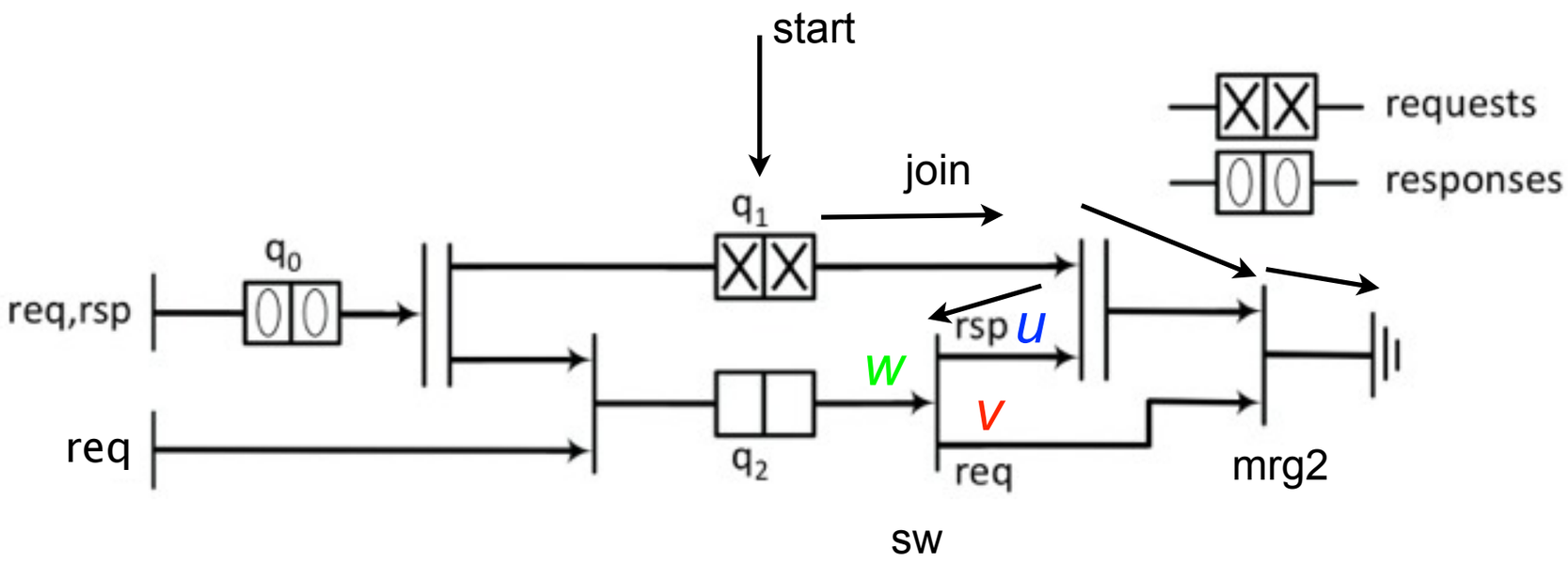
Block(u) = **Block**(w)



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

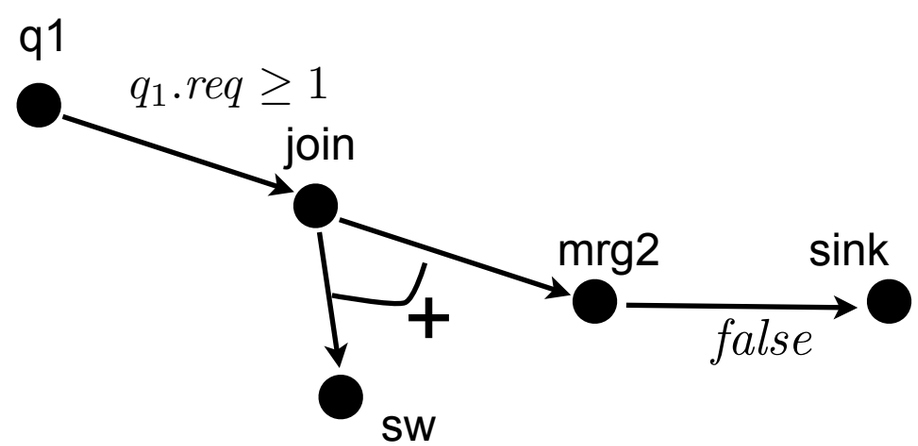
we assume fair sinks

Step 2 / labelled dependency graph (2)

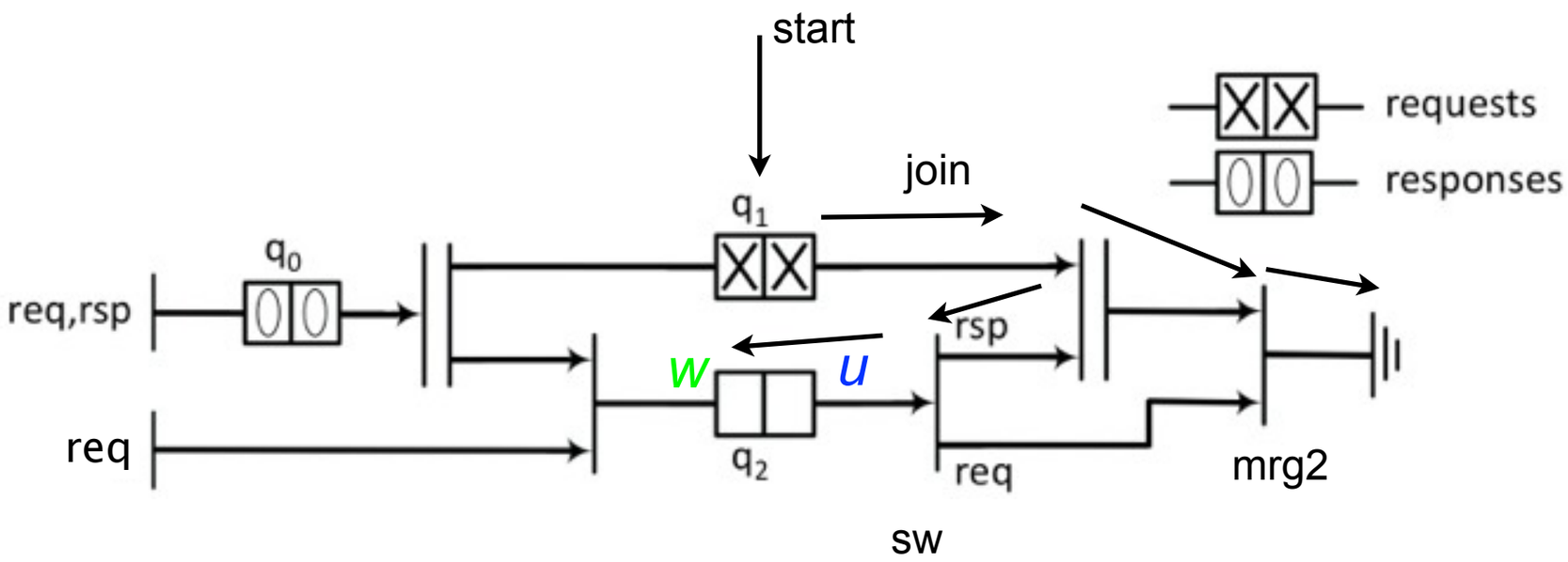


Idle(u) = **Idle**(w)

backwards to the switch



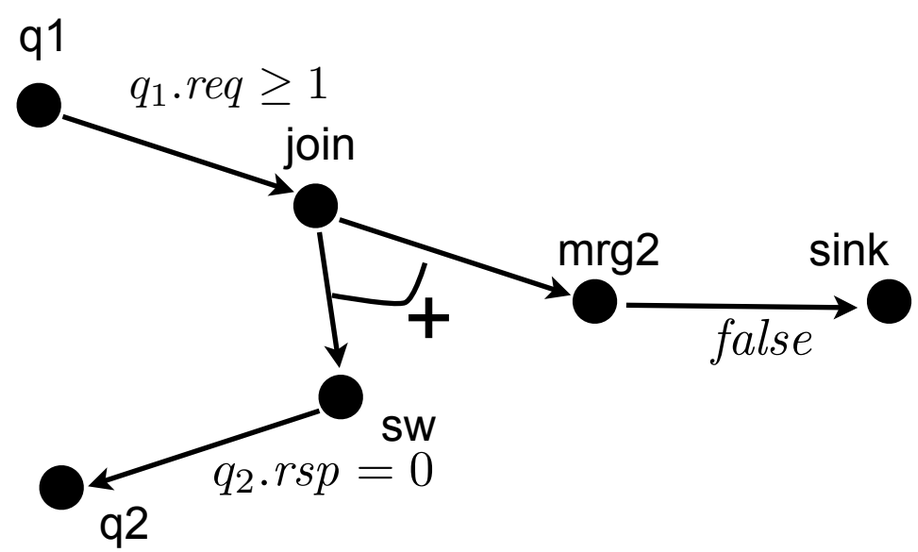
Step 2 / labelled dependency graph (2)



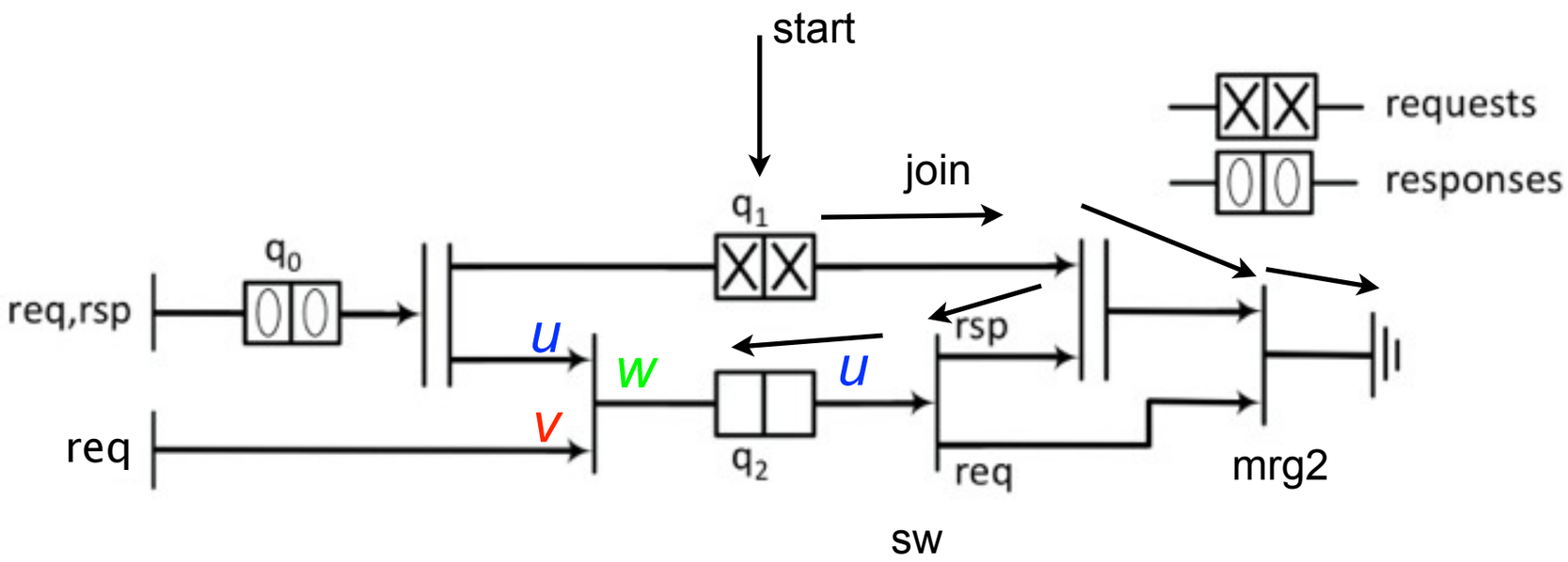
Idle(u) = **Idle**(w) . **Empty**(q2)

backwards to the queue

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



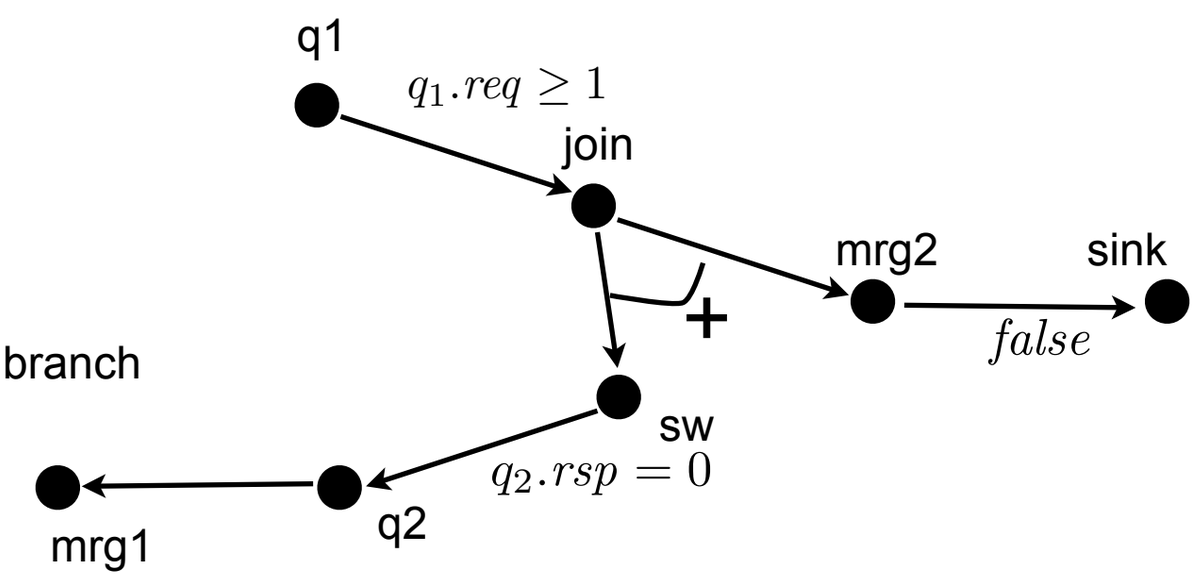
Step 2 / labelled dependency graph (2)



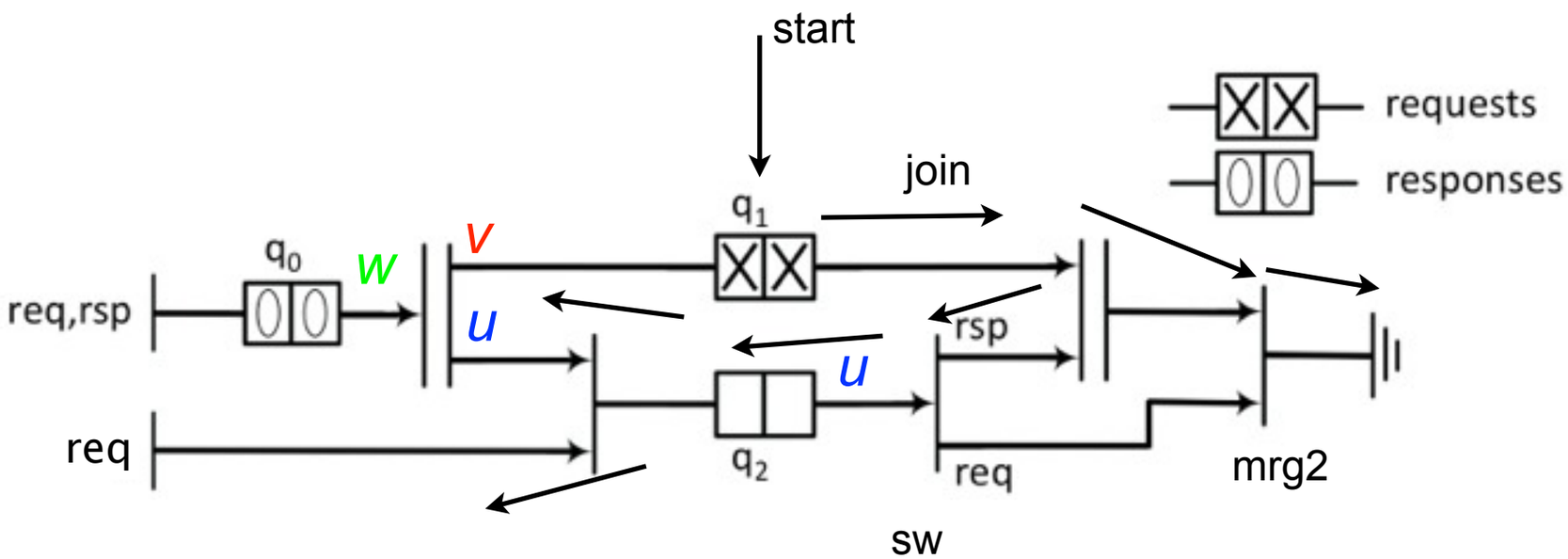
Idle(w) = **Idle**(u) . **Idle**(v)

backwards to the merge and branch

note branching is bad for us



Step 2 / labelled dependency graph (2)

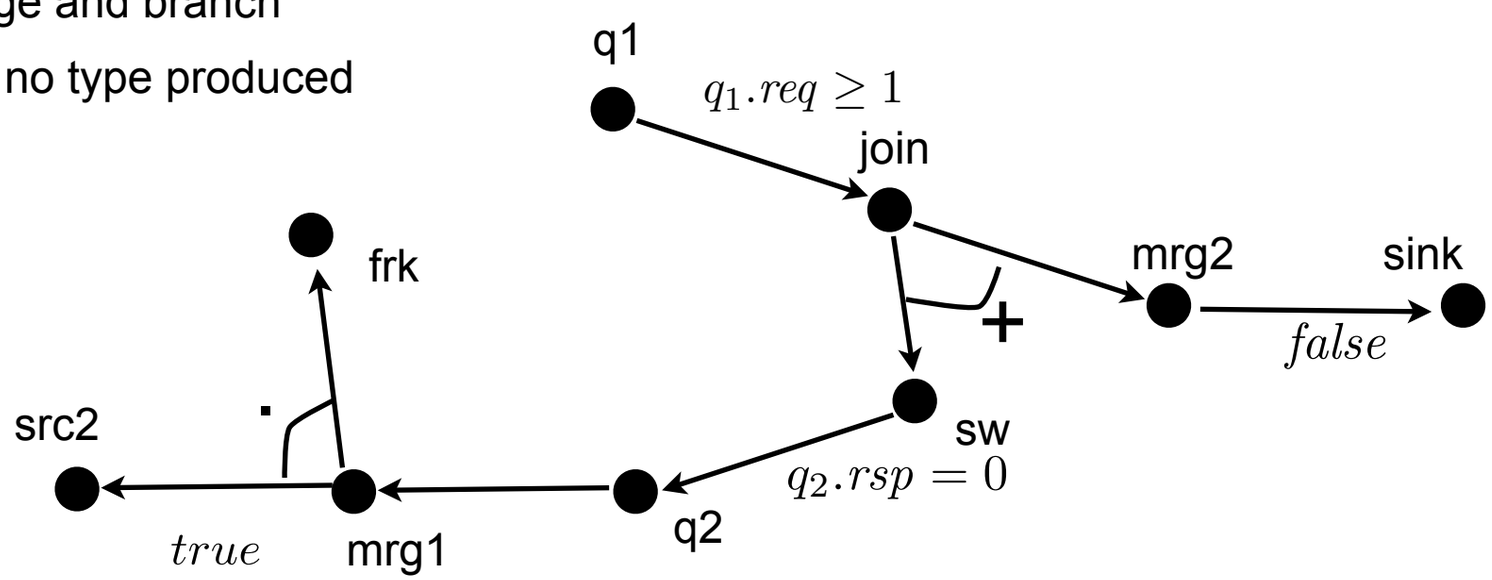


Idle(u) = **Block**(v) + **Idle**(w)

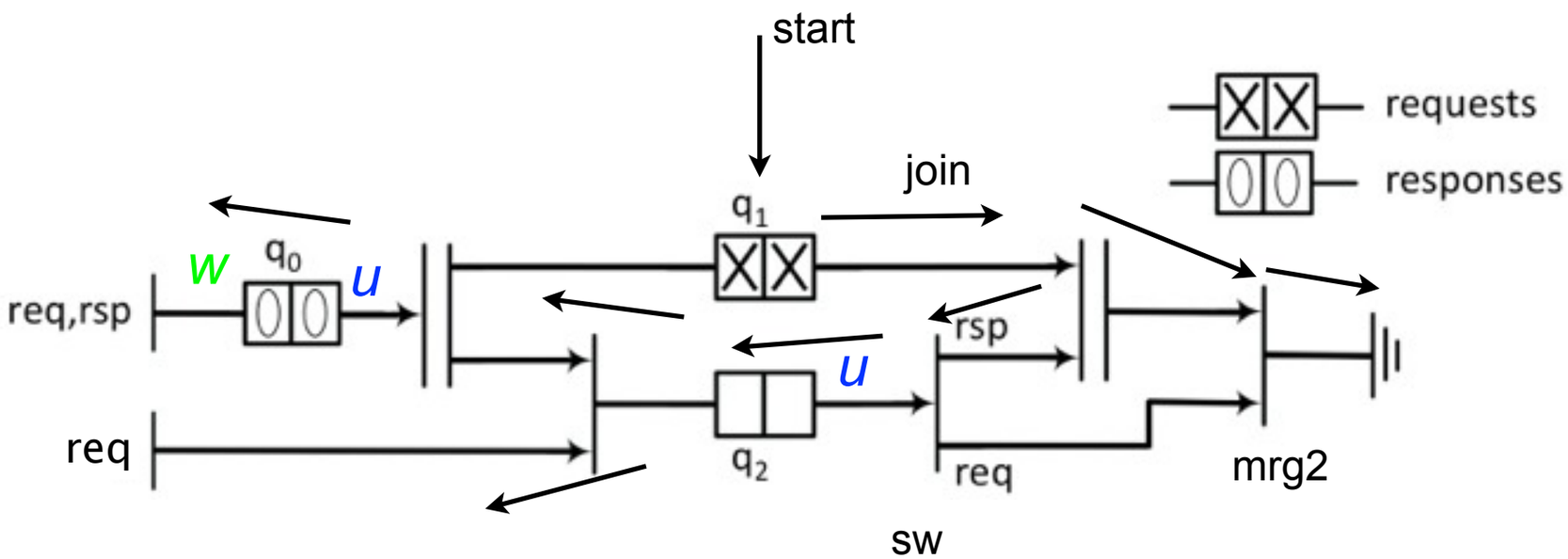
backwards to the merge and branch

to the source - idle if no type produced

to the fork

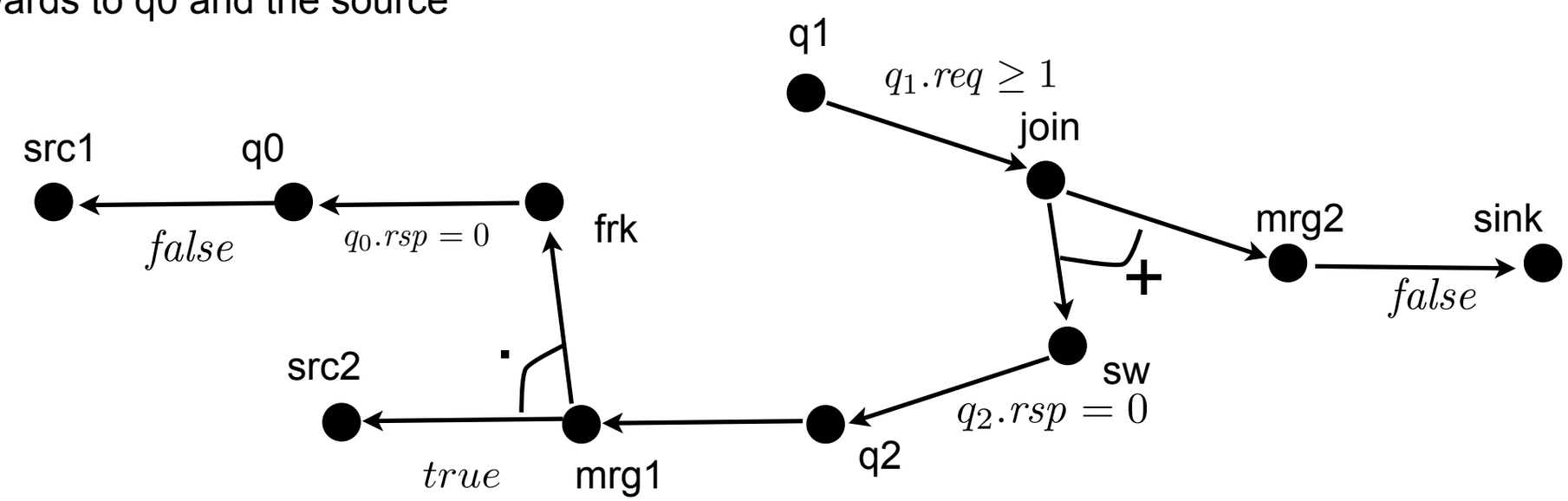


Step 2 / labelled dependency graph (2)

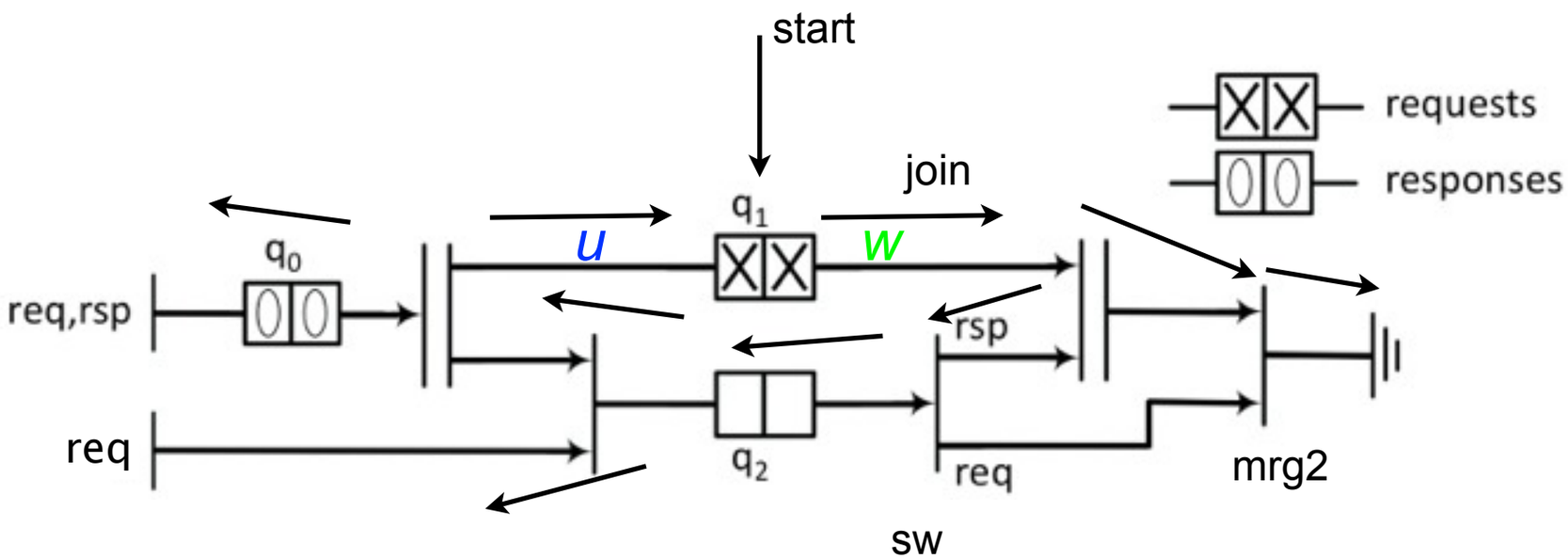


Idle(u) = **Idle**(w) . **Empty**(q0)

backwards to q0 and the source

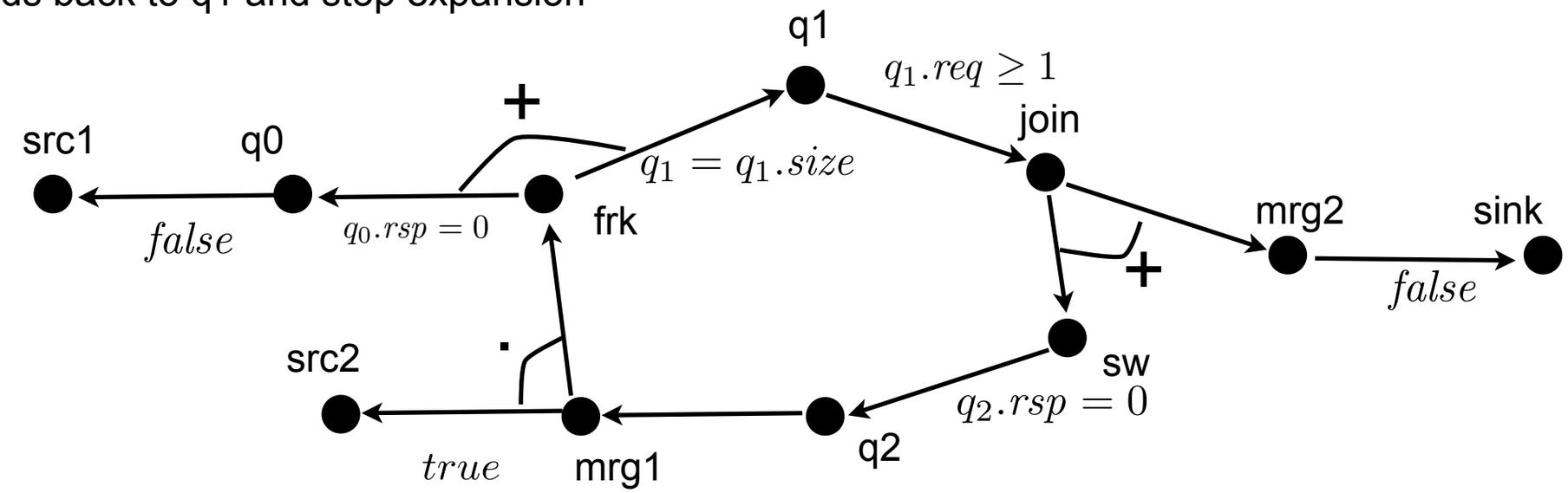


Step 2 / labelled dependency graph (2)



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

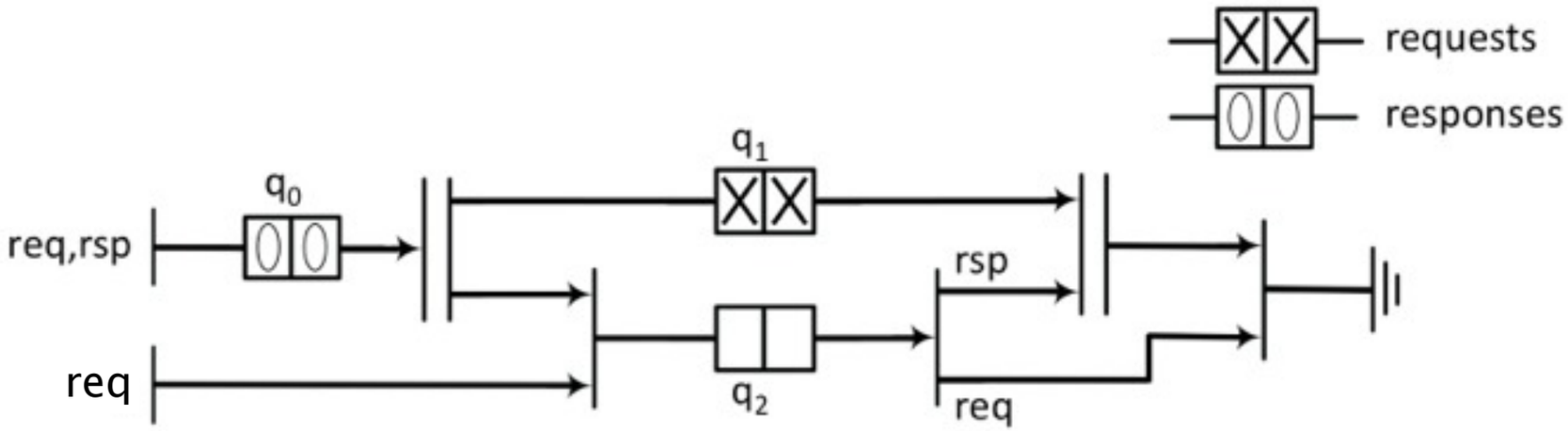
forwards back to q1 and stop expansion

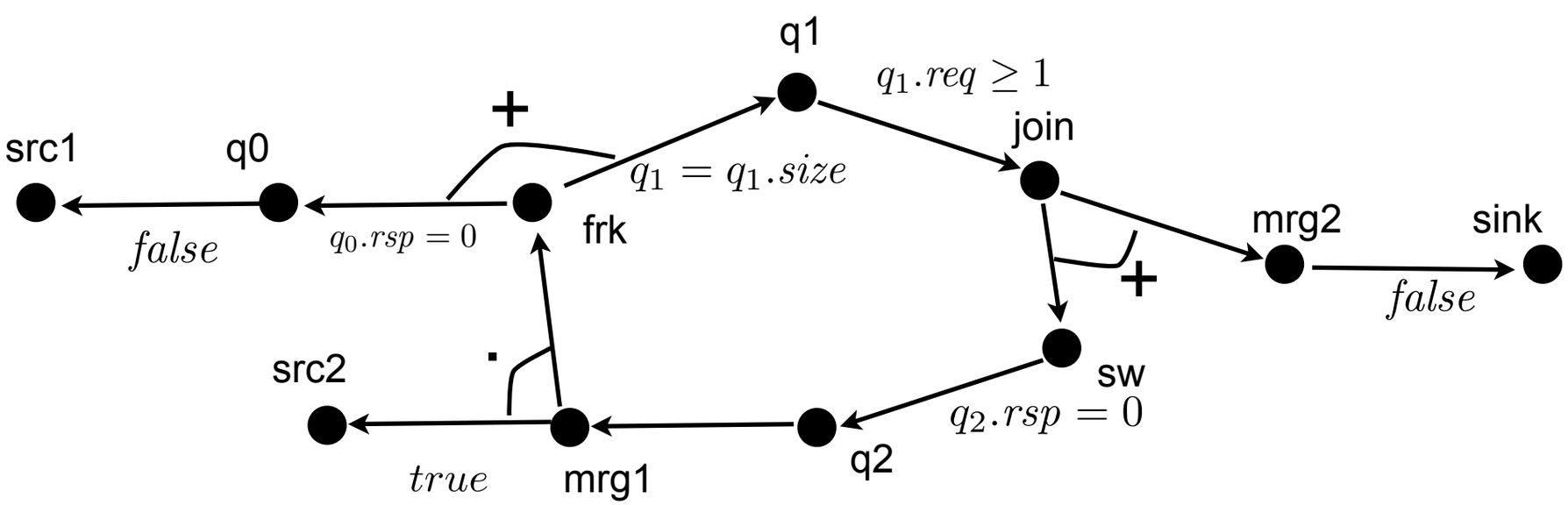


General approach for deadlock detection in xMAS networks

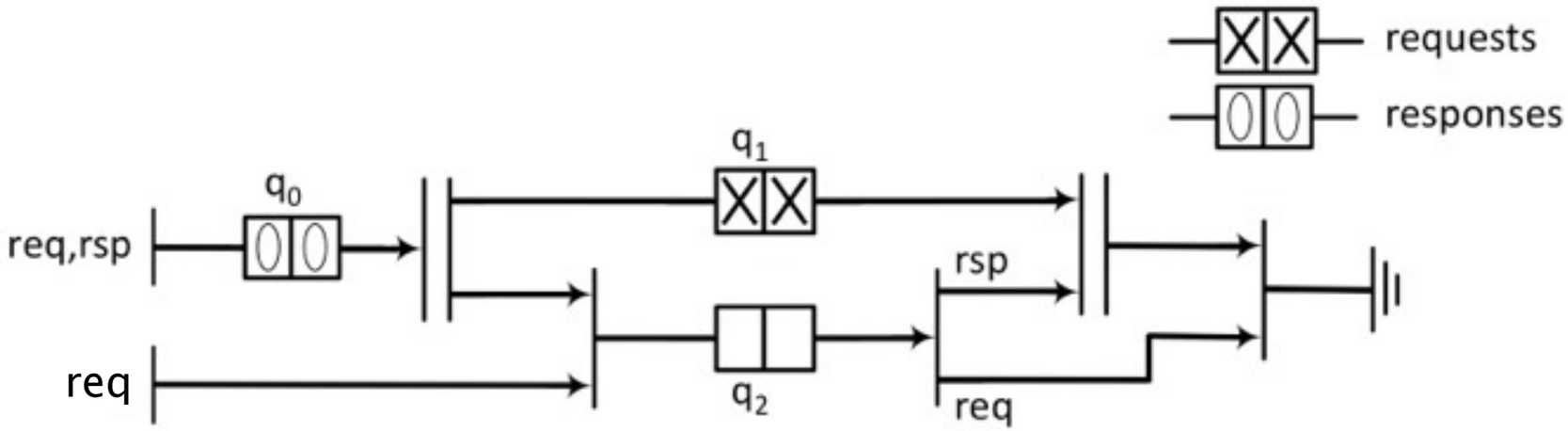
- Define Blocking Equations for all components
  - Equations capture the reason why a component is idle or blocking
- Build a labelled waiting graph for each queue
  - Labels correspond to the equations
  - Graph captures the topology, i.e., the dependencies between the xMAS components
- Search for a feasible logically closed subgraph
  - Corresponds to a deadlock situation
  - Feasibility checked using Linear Programming

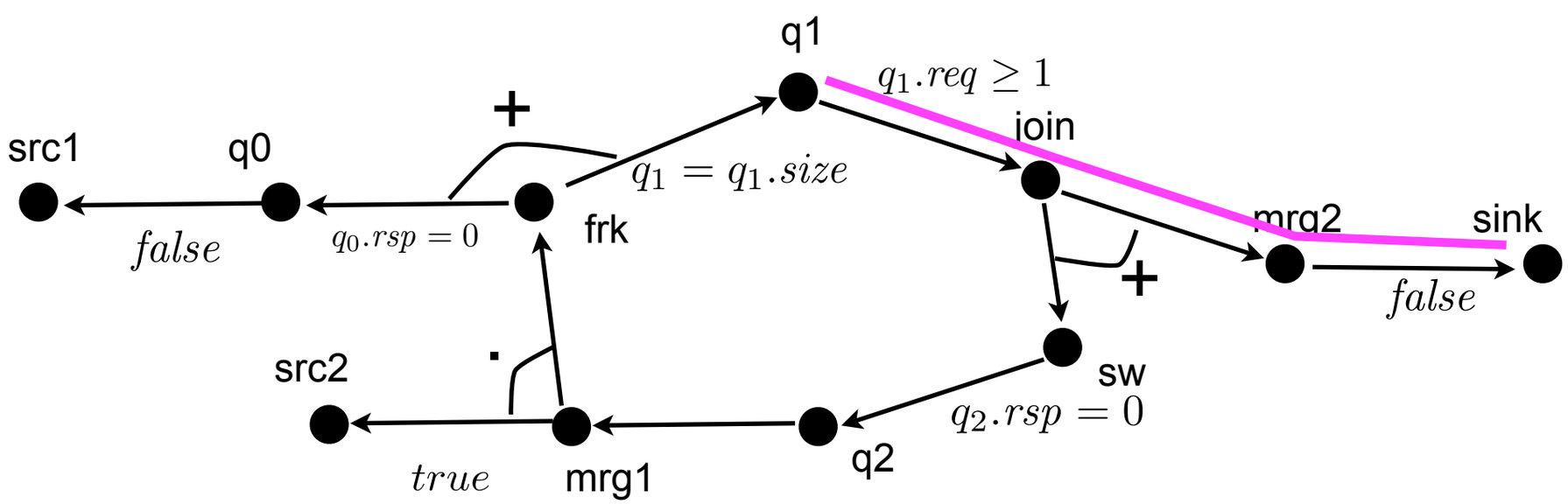
Step 2 / logically closed subgraph 1



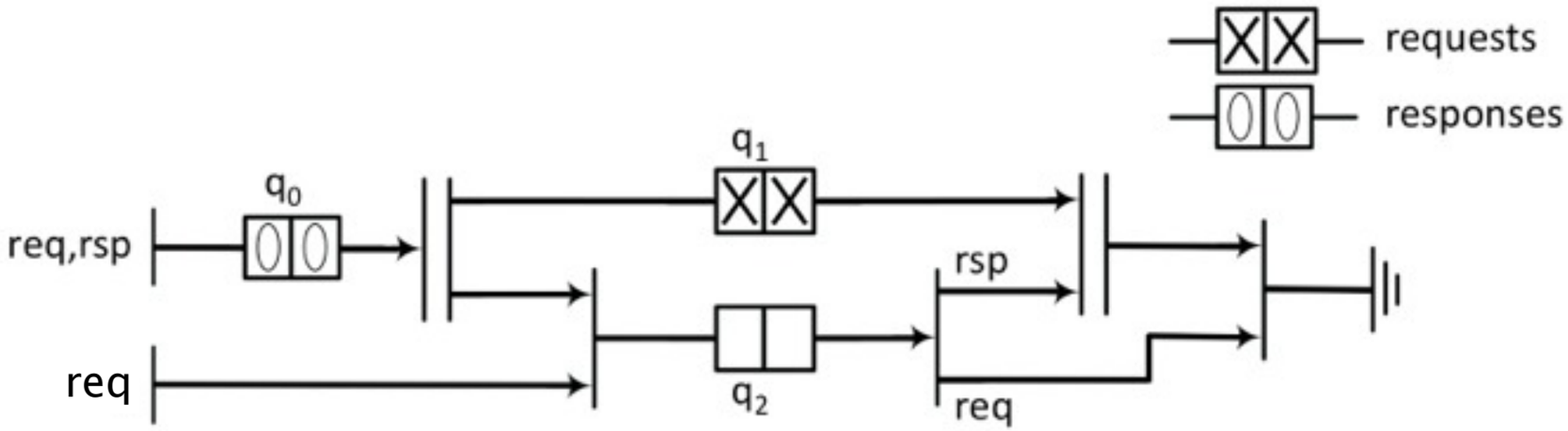


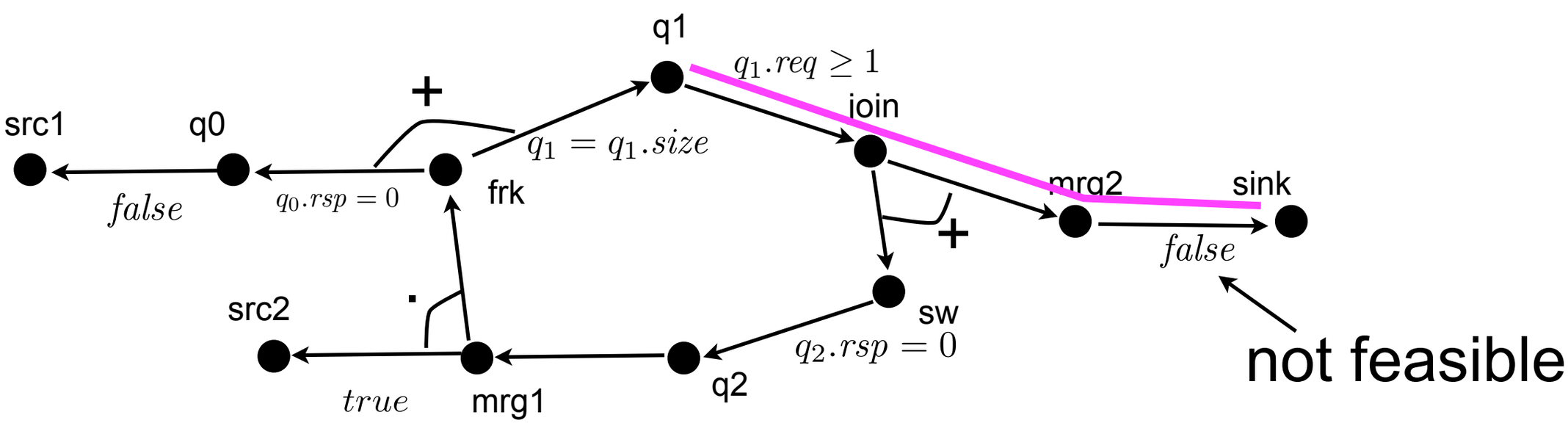
Step 2 / logically closed subgraph 1



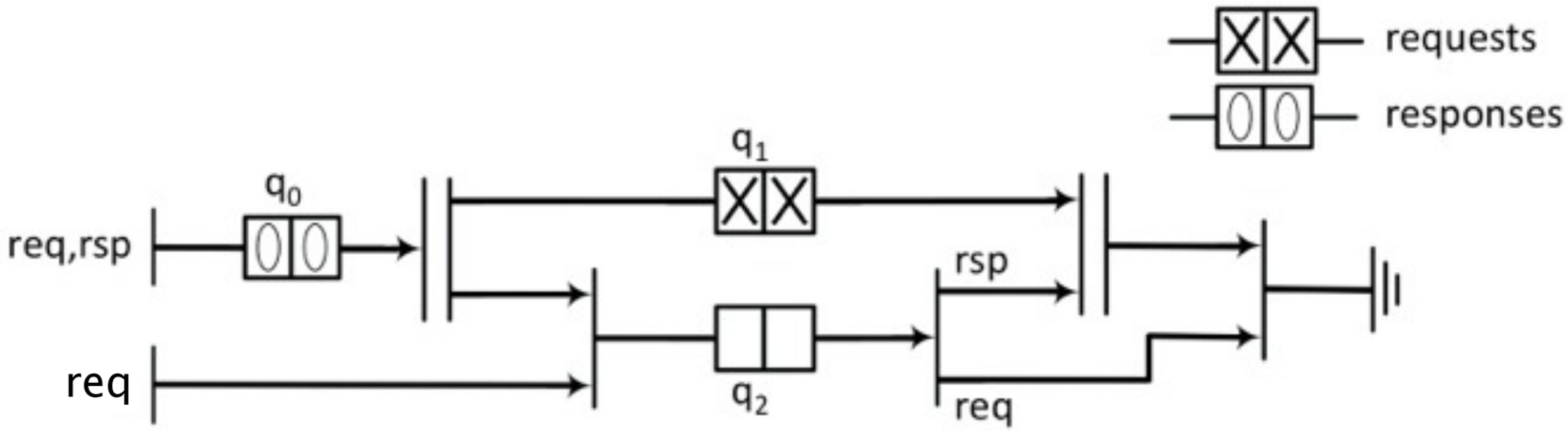


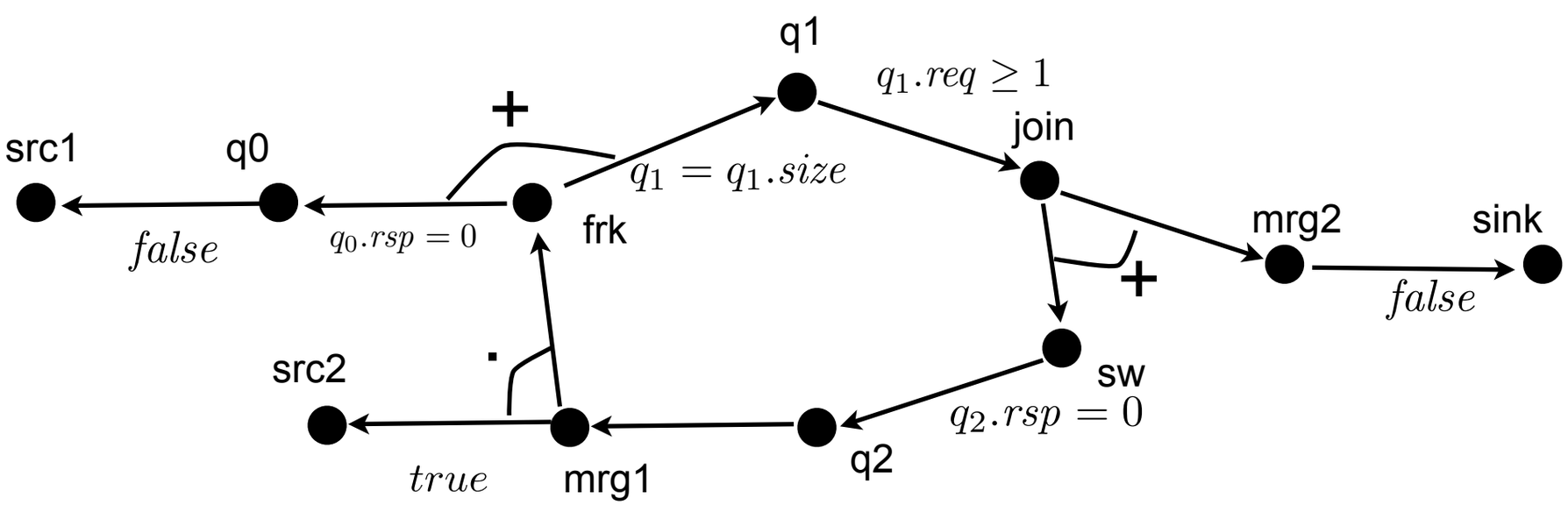
Step 2 / logically closed subgraph 1



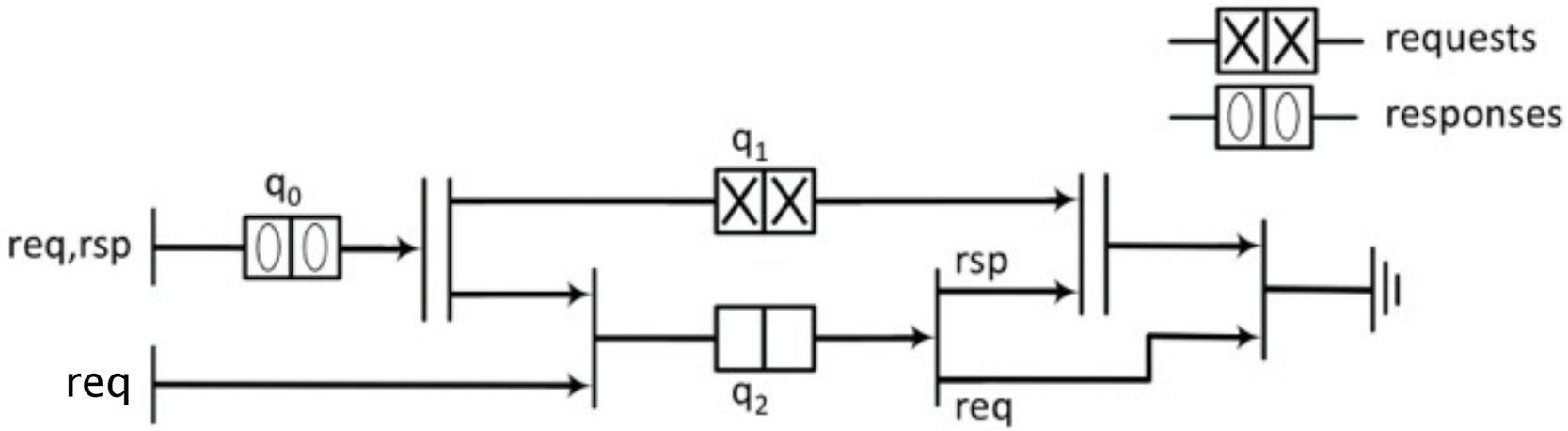


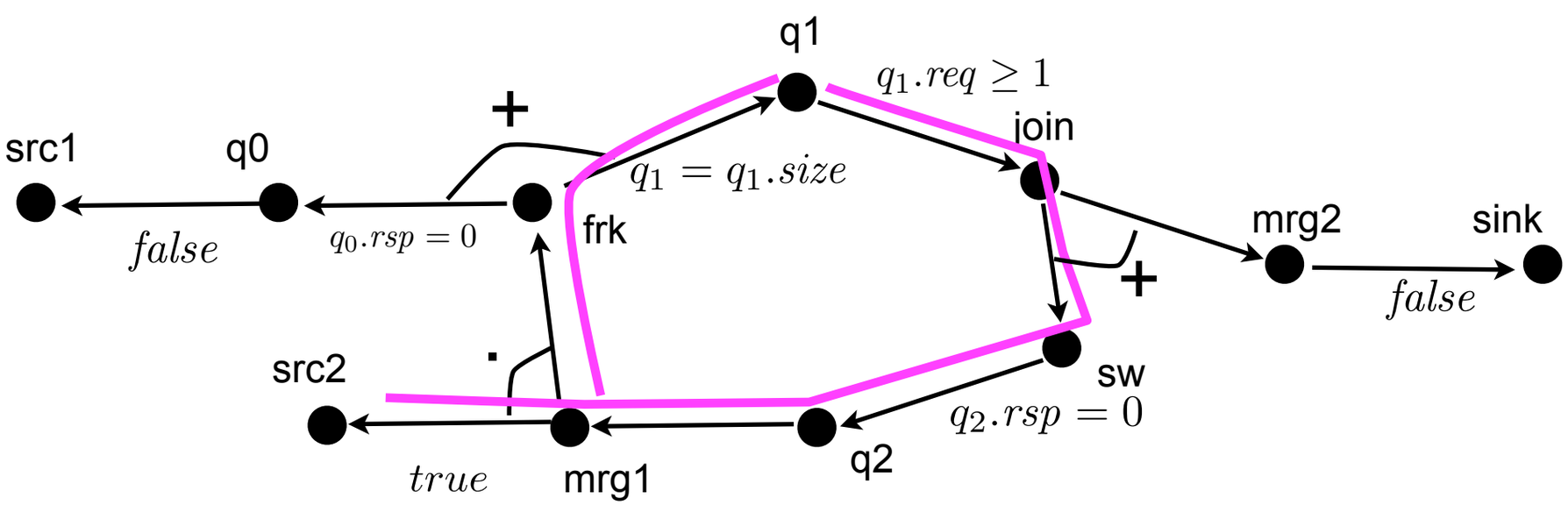
Step 2 / logically closed subgraph 2



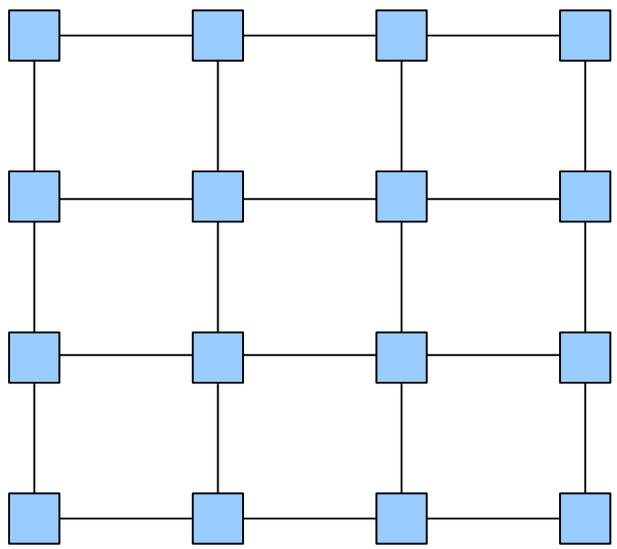


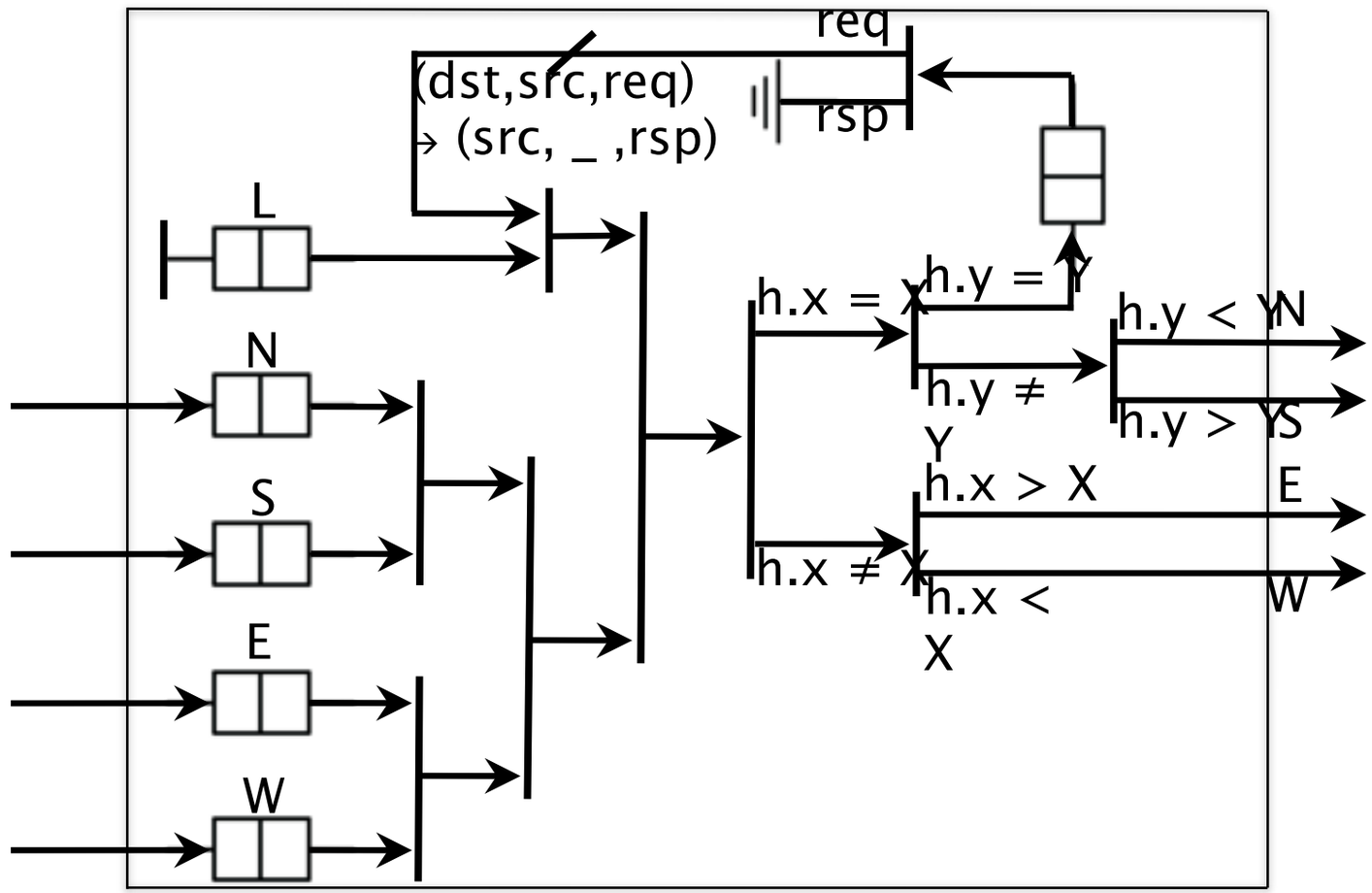
Step 2 / logically closed subgraph 2





Experimental Results

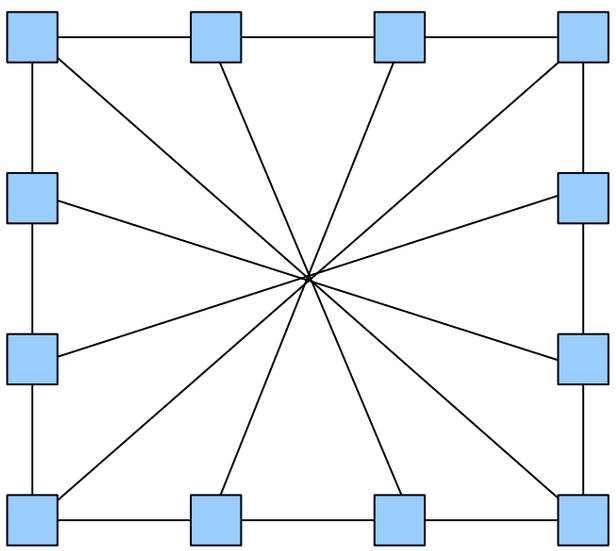


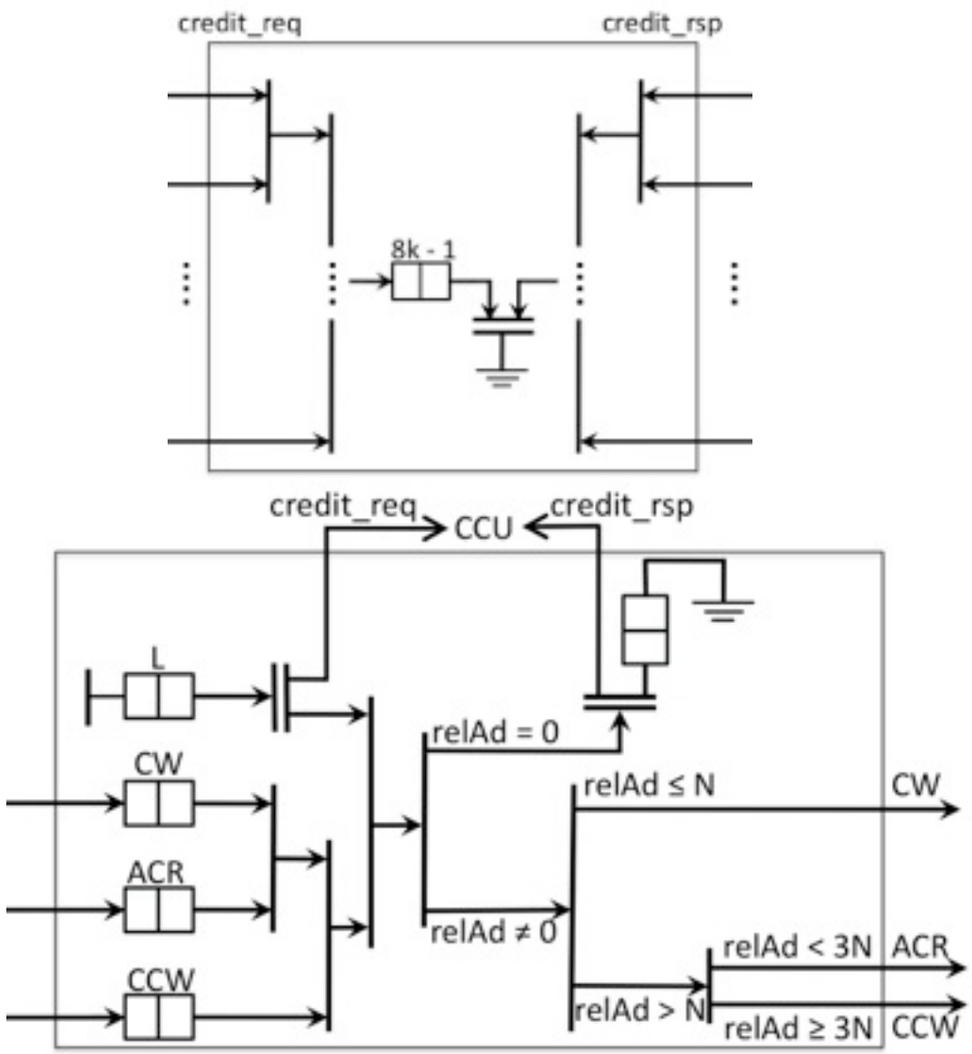


With deadlocks: a 14x14 mesh with 3724 components in 6.05 seconds

Without deadlocks: a 14x14 mesh with 3724 components in 1.31 seconds

Experimental Results





With deadlocks: a 28 ring with 477 components in 0.5 seconds
Without deadlocks: a 28 ring with 477 components in 6.6 seconds

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

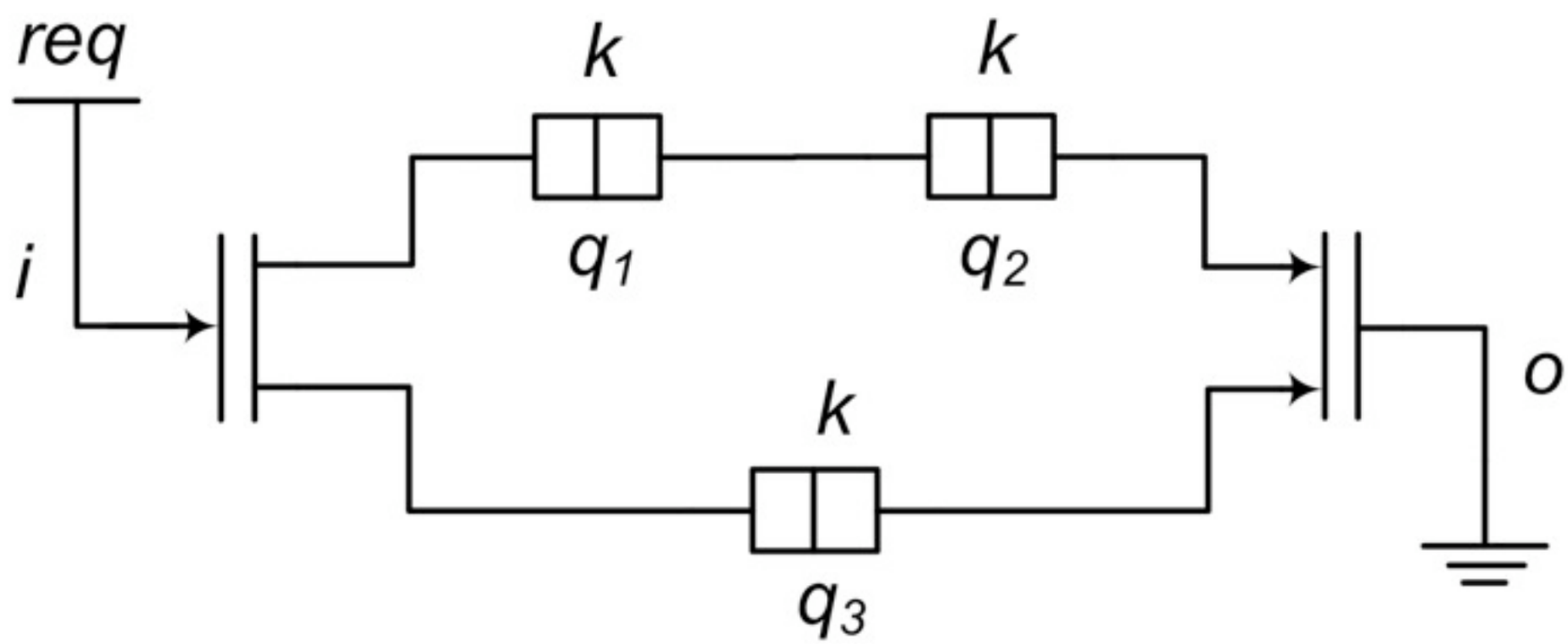
- Tool to detect message dependent deadlocks
  - Expressive language for routing, protocol, injection, etc.
  - Intricate deadlocks
  - Very efficient due to equations
  - Necessary and sufficient for structural deadlocks
  - Counterexamples

- Future work:
  - Still need to be formally proven
  - Composition/Hierarchy
    - Check sub-networks first and then compose

Thanks !

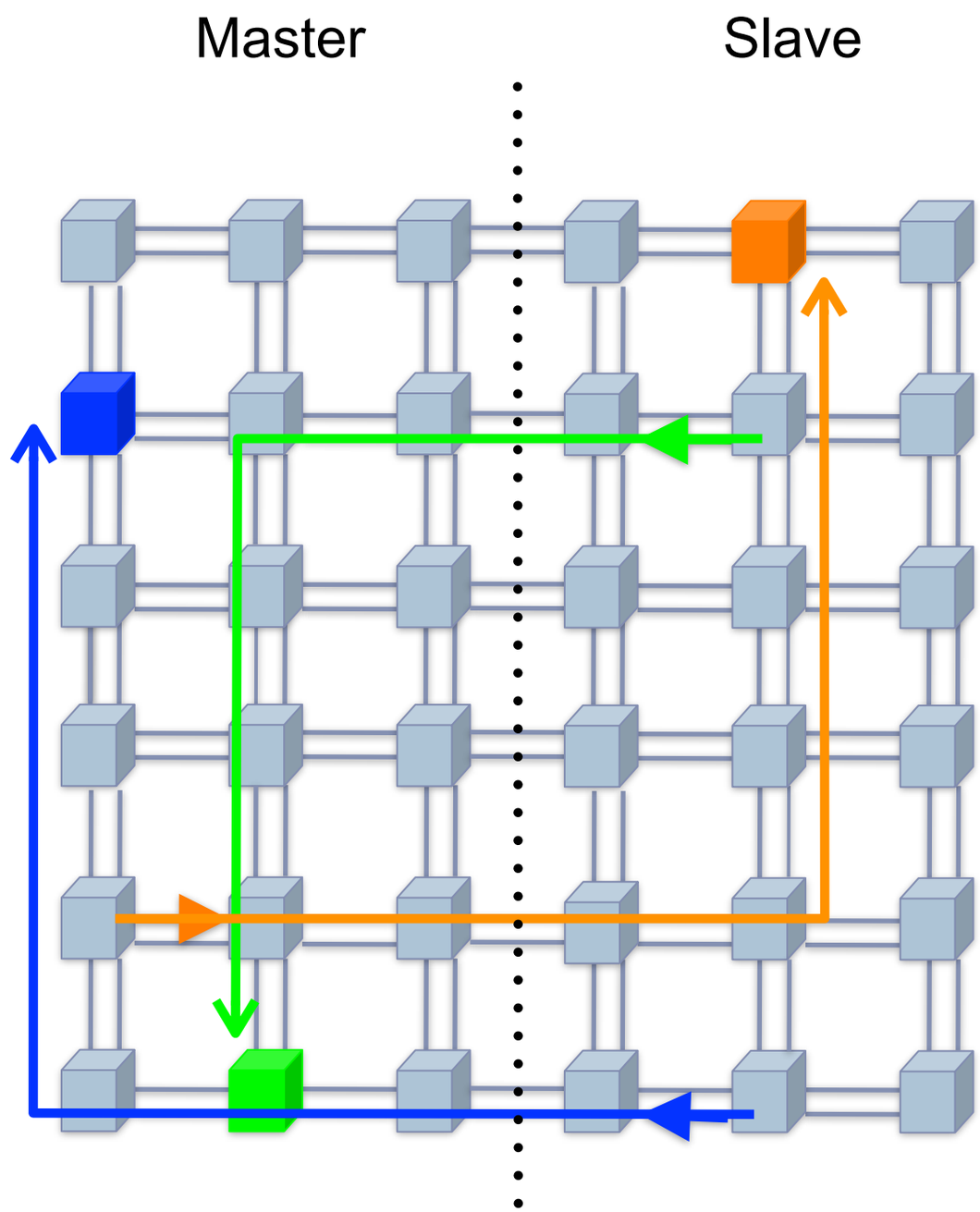
Deadlock example 3

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



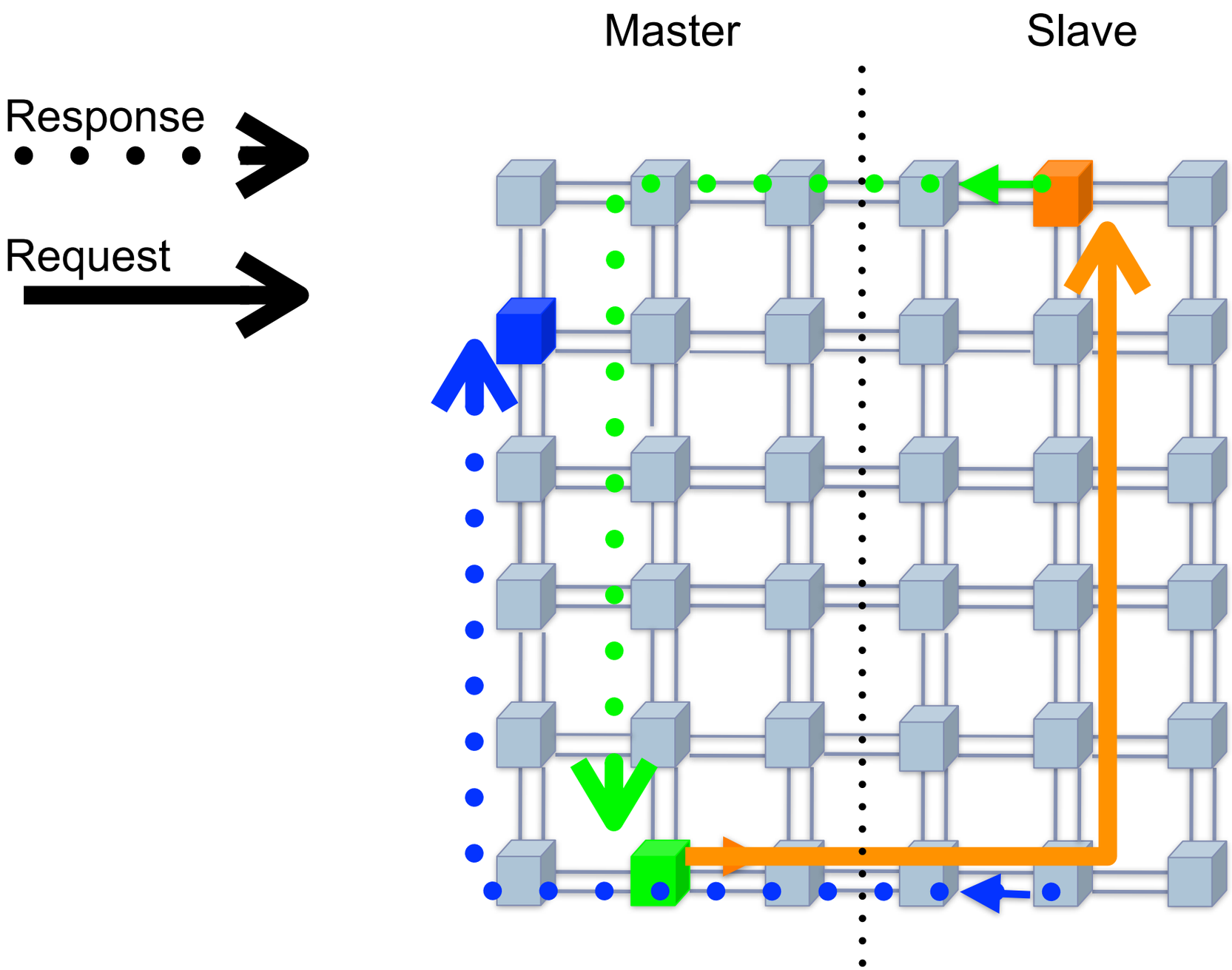
Networks-on-Chips: Example 1

Core distribution:



- Masters on the right/slaves on the left

Networks-on-Chips: Example 1



- The system is deadlock-free!