Naturalized Communication and Testing

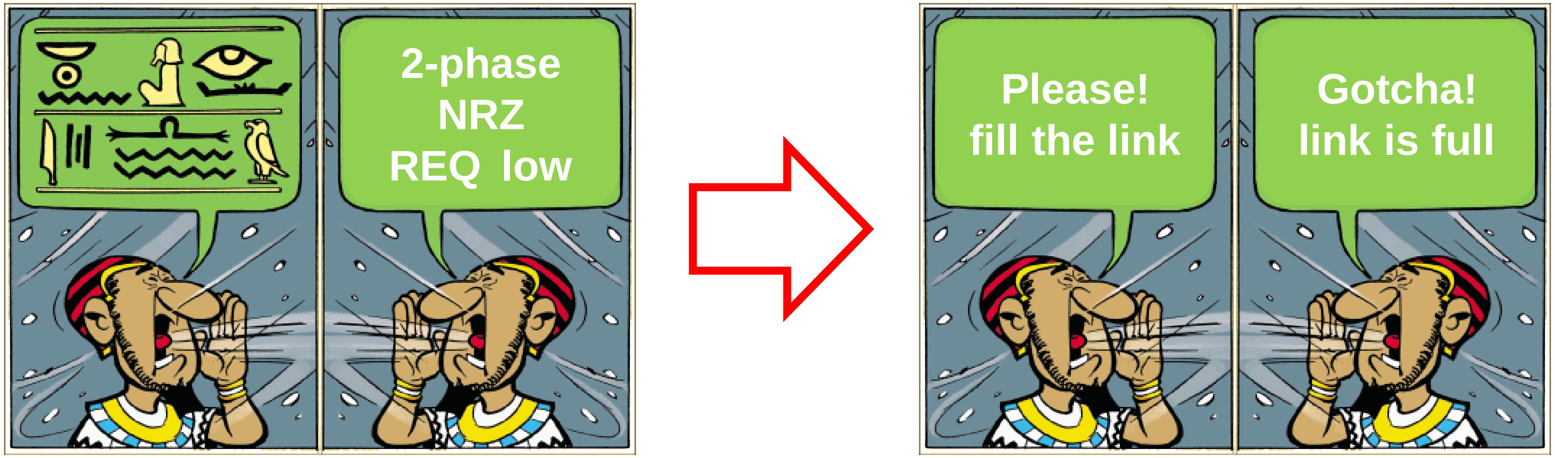
Marly Roncken
Swetha Mettala Gilla
Hoon Park
Navaneeth Jamadagni
Chris Cowan
Ivan Sutherland

Asynchronous Research Center
Portland State University
ASYNC 2015, 3-6 May

Outline

PART 1: naturalized communication

- exposes the fundamental pipeline actions underlying all handshakes
- to obtain a standard protocol interface for translation-free communication
- to simplify the exchange of designs + tools

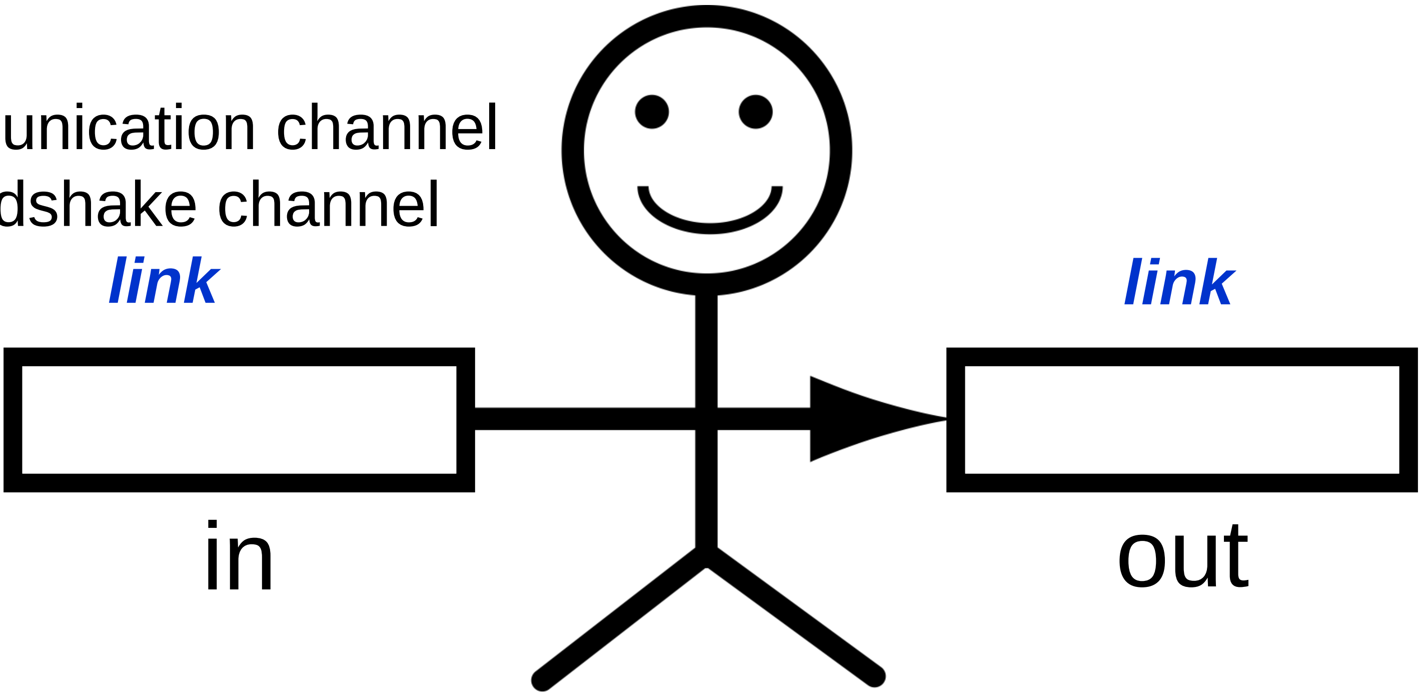


PART 2: naturalized testing

- emphasizes the role of actions
- by using dedicated action control: MrGO
- to safely start, stop, and freeze actions individually
- for single-step, multi-step, and at-speed test + debug

PART 1
naturalized communication

Dataflow pipeline: building blocks



handshake component
module
joint

communication channel
handshake channel
link

link

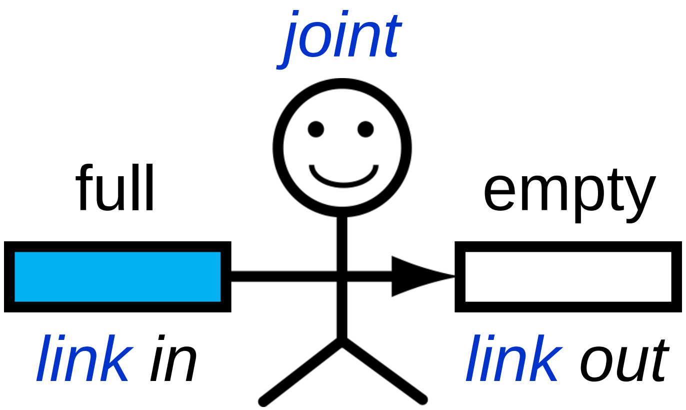
Dataflow pipeline: action

WHEN to act:

in is full

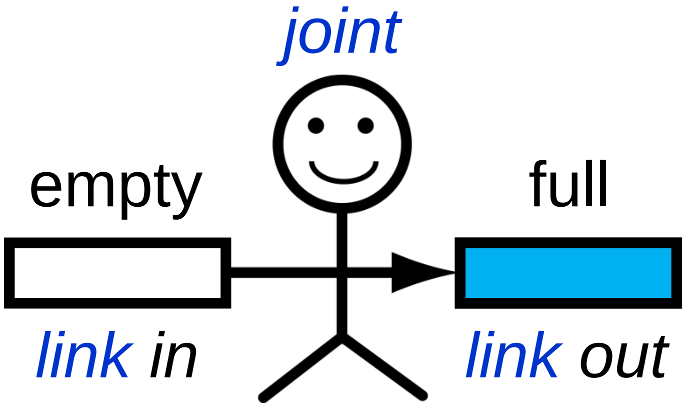
and

out is empty

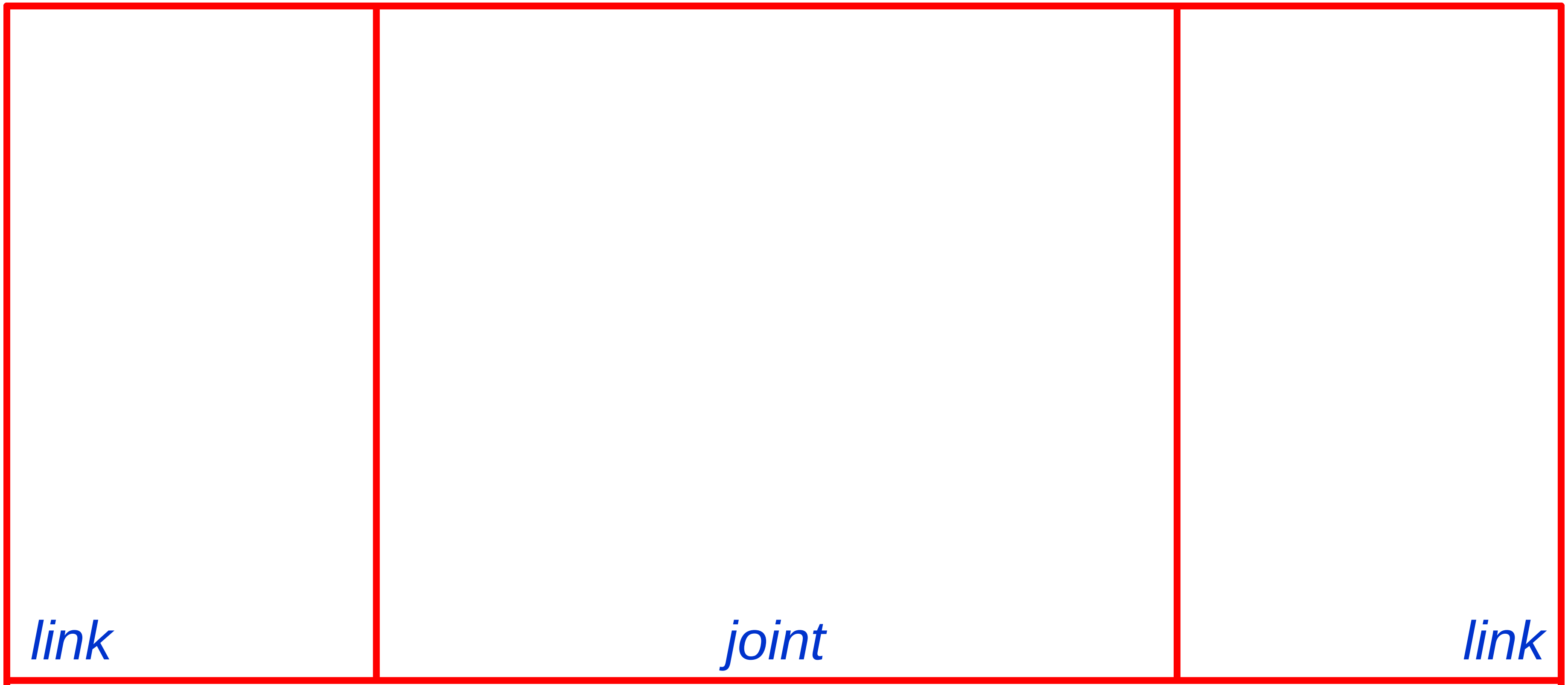


WHAT to do:

- copy data
- drain *in*
- fill *out*



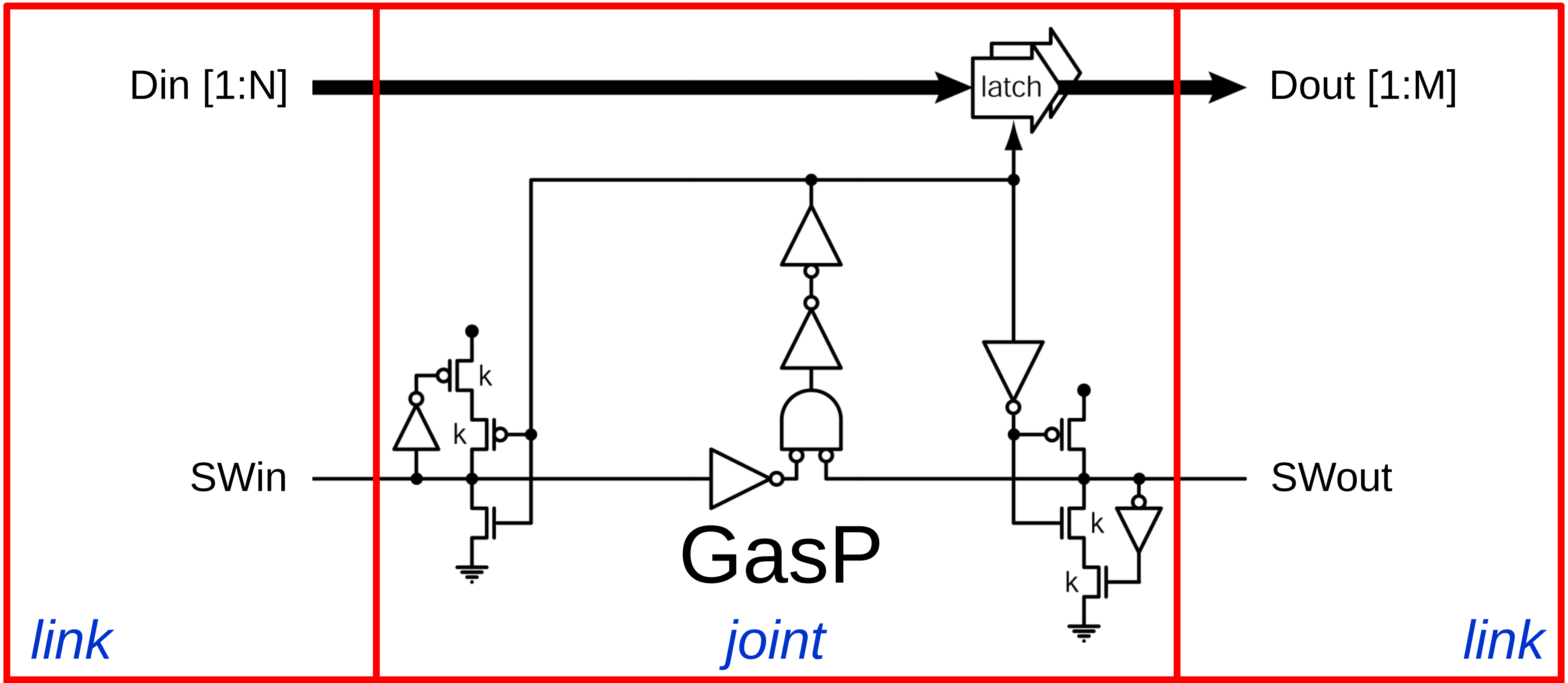
Dataflow pipeline: original designs



Drawbacks:

- link has wires only
- joint has all the computation + communication logic
- link-joint interface changes per handshake protocol

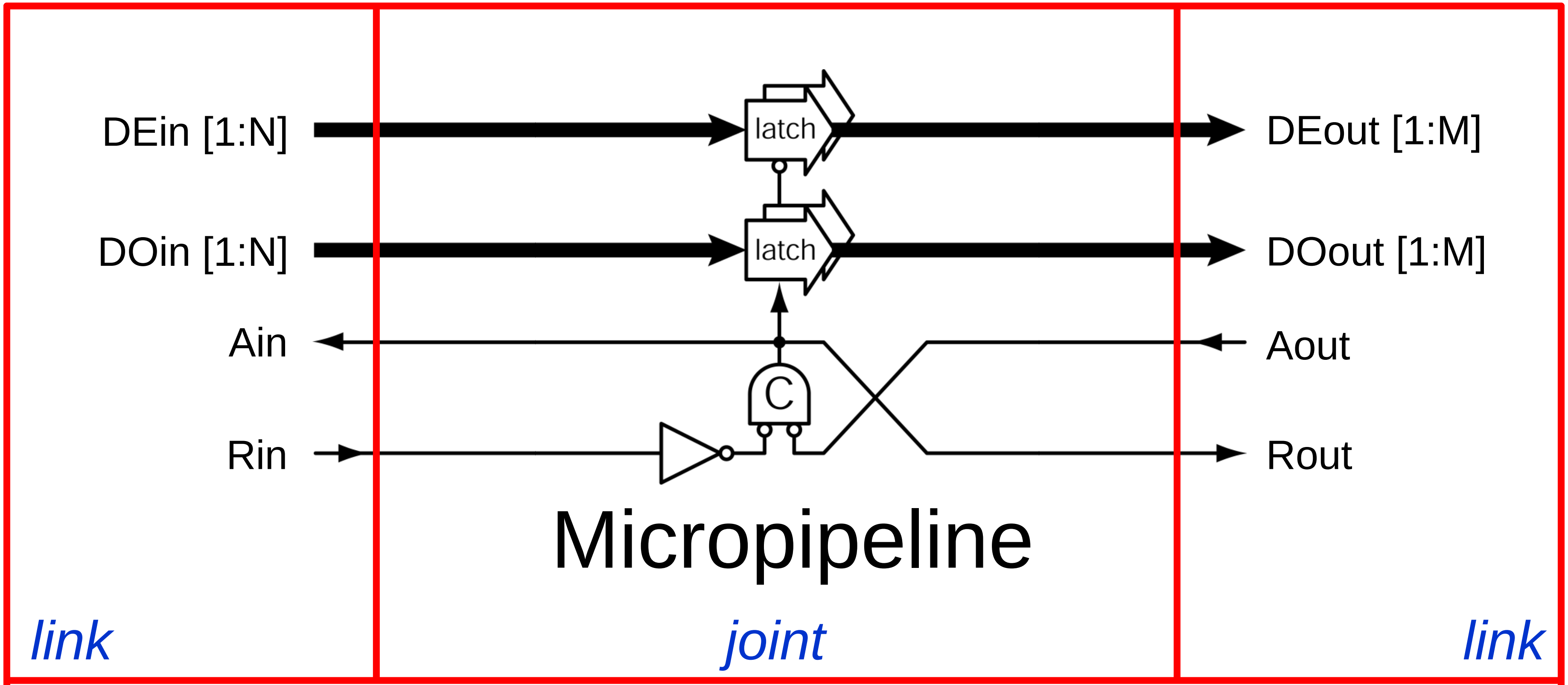
Dataflow pipeline: original designs



Drawbacks:

- link has wires only
- joint has all the computation + communication logic
- link-joint interface changes per handshake protocol

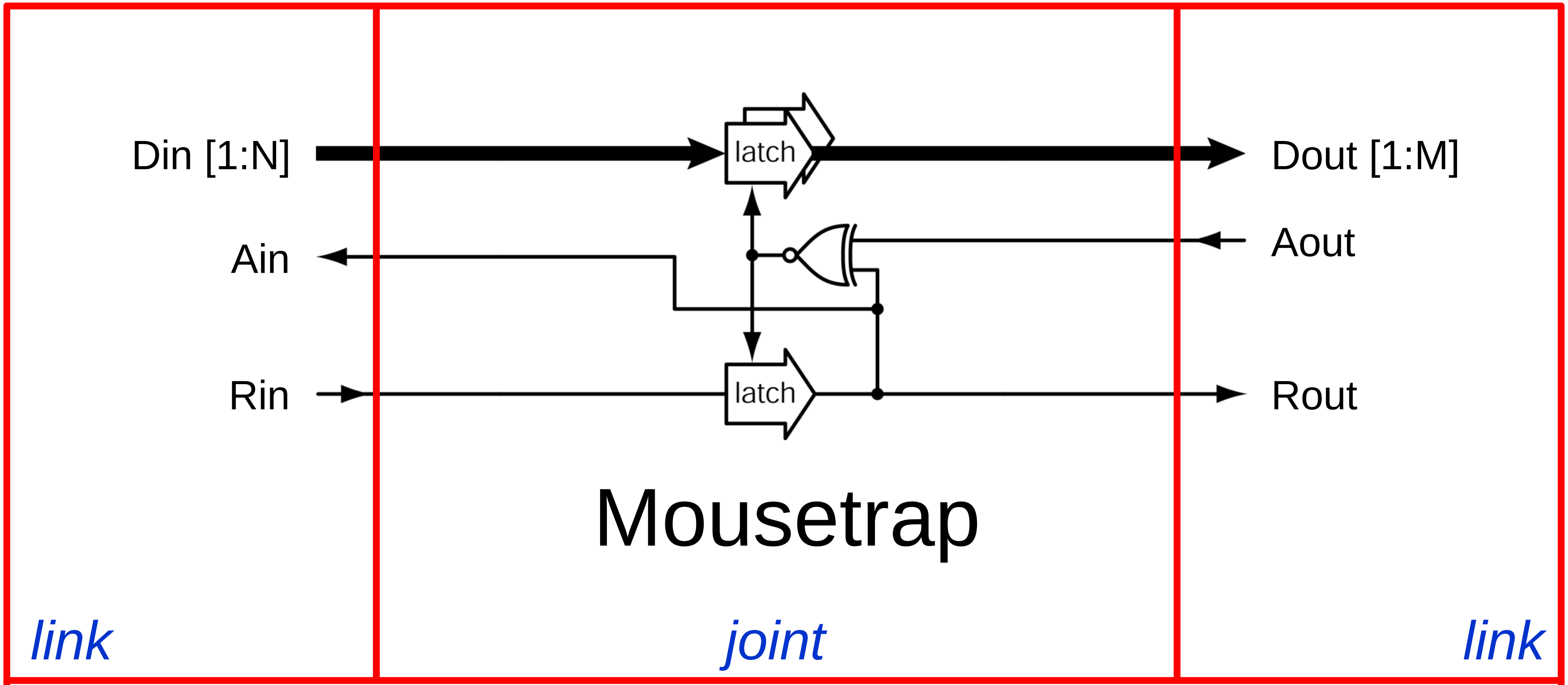
Dataflow pipeline: original designs



Drawbacks:

- link has wires only
- joint has all the computation + communication logic
- link-joint interface changes per handshake protocol

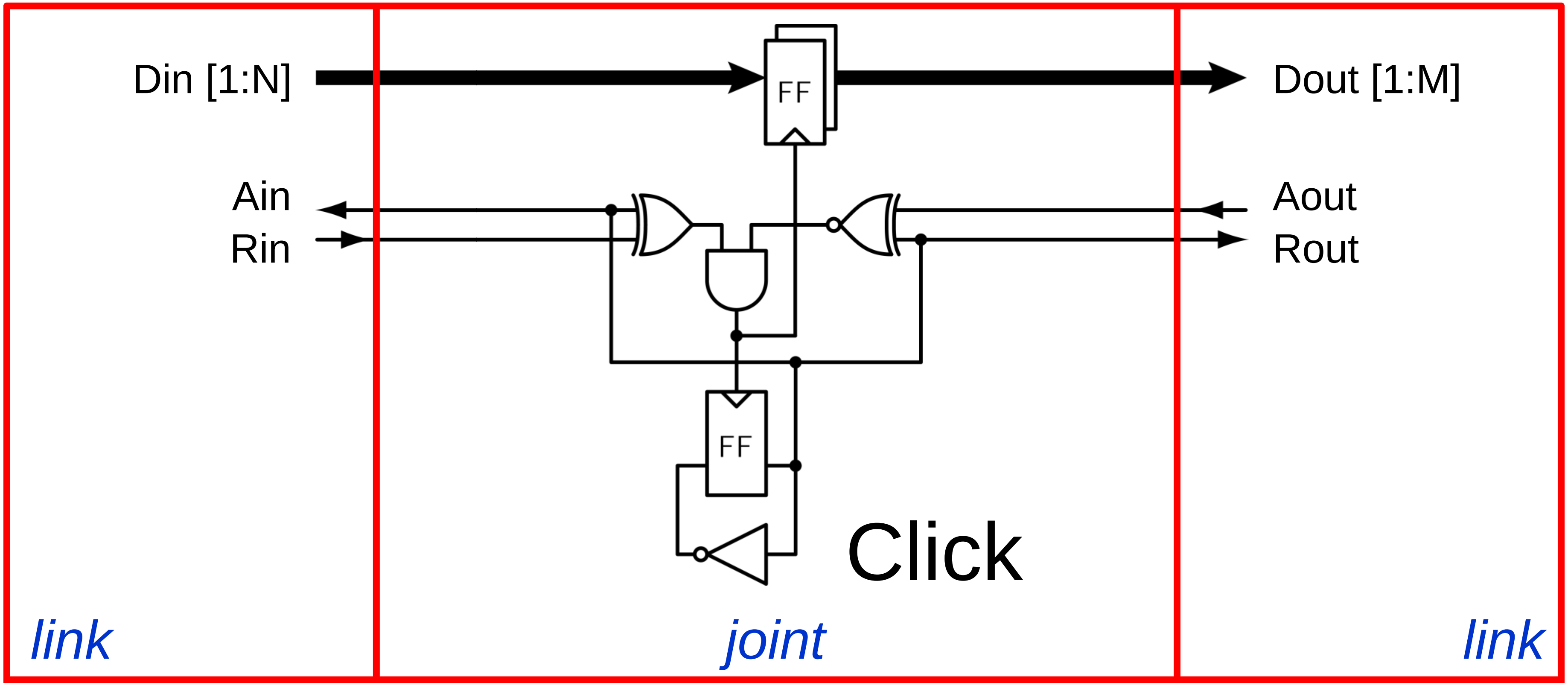
Dataflow pipeline: original designs



Drawbacks:

- link has wires only
- joint has all the computation + communication logic
- link-joint interface changes per handshake protocol

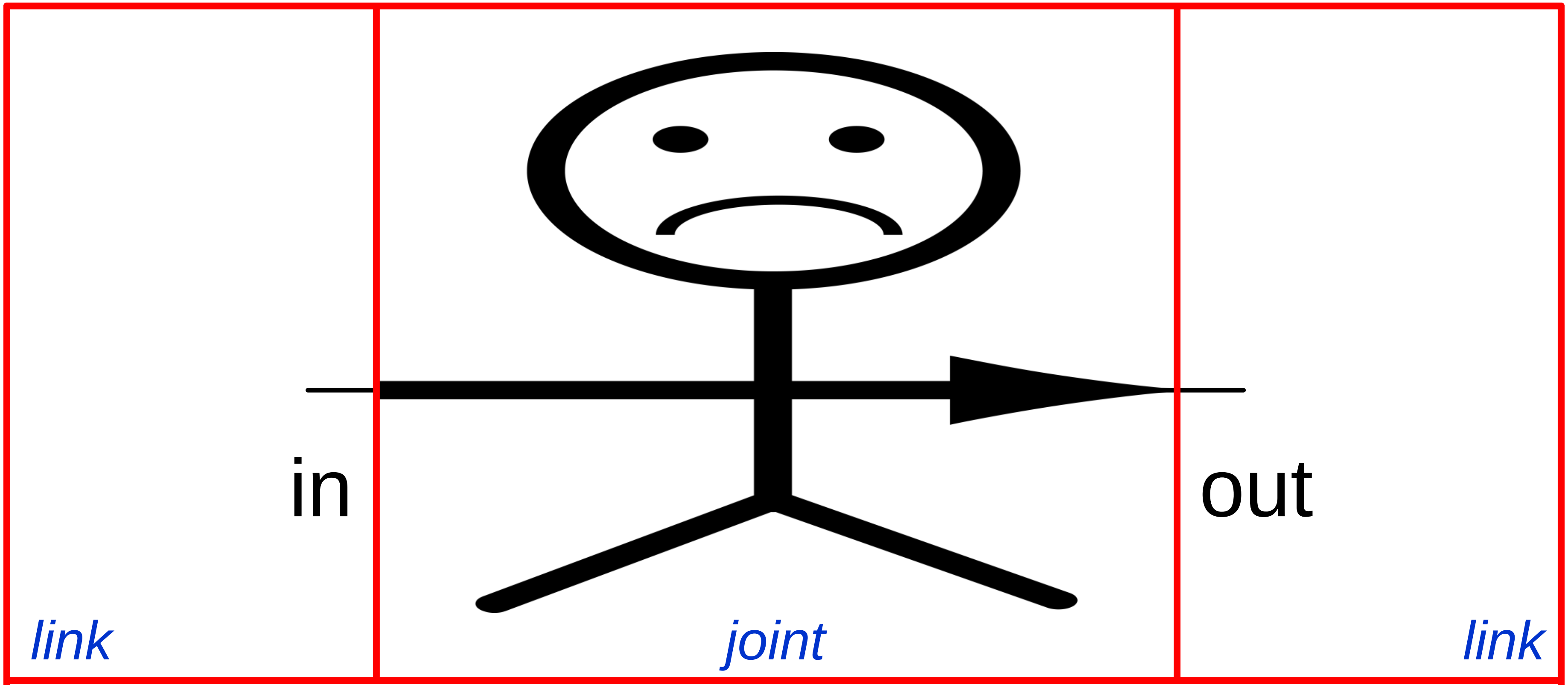
Dataflow pipeline: original designs



Drawbacks:

- link has wires only
- joint has all the computation + communication logic
- link-joint interface changes per handshake protocol

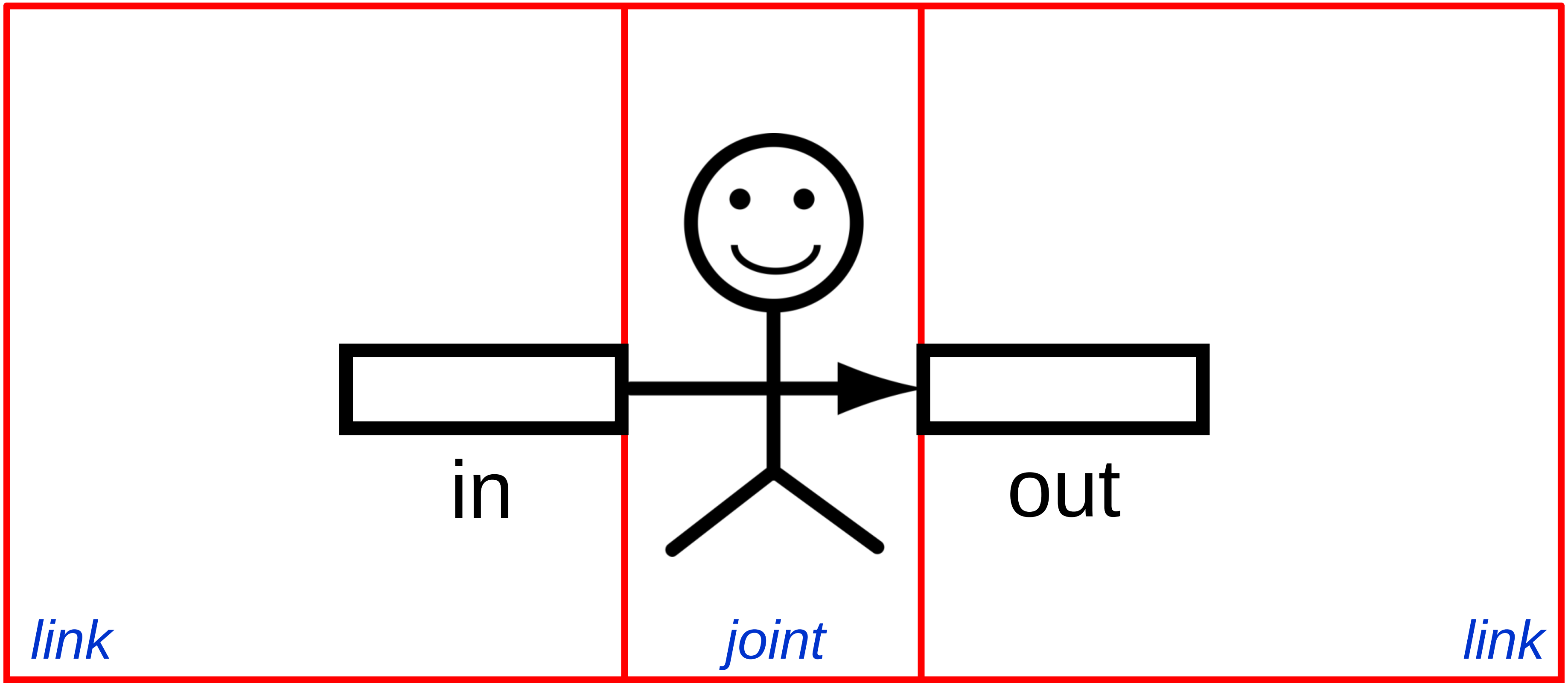
Dataflow pipeline: original designs



Drawbacks:

the joint is too fat and the links are too thin

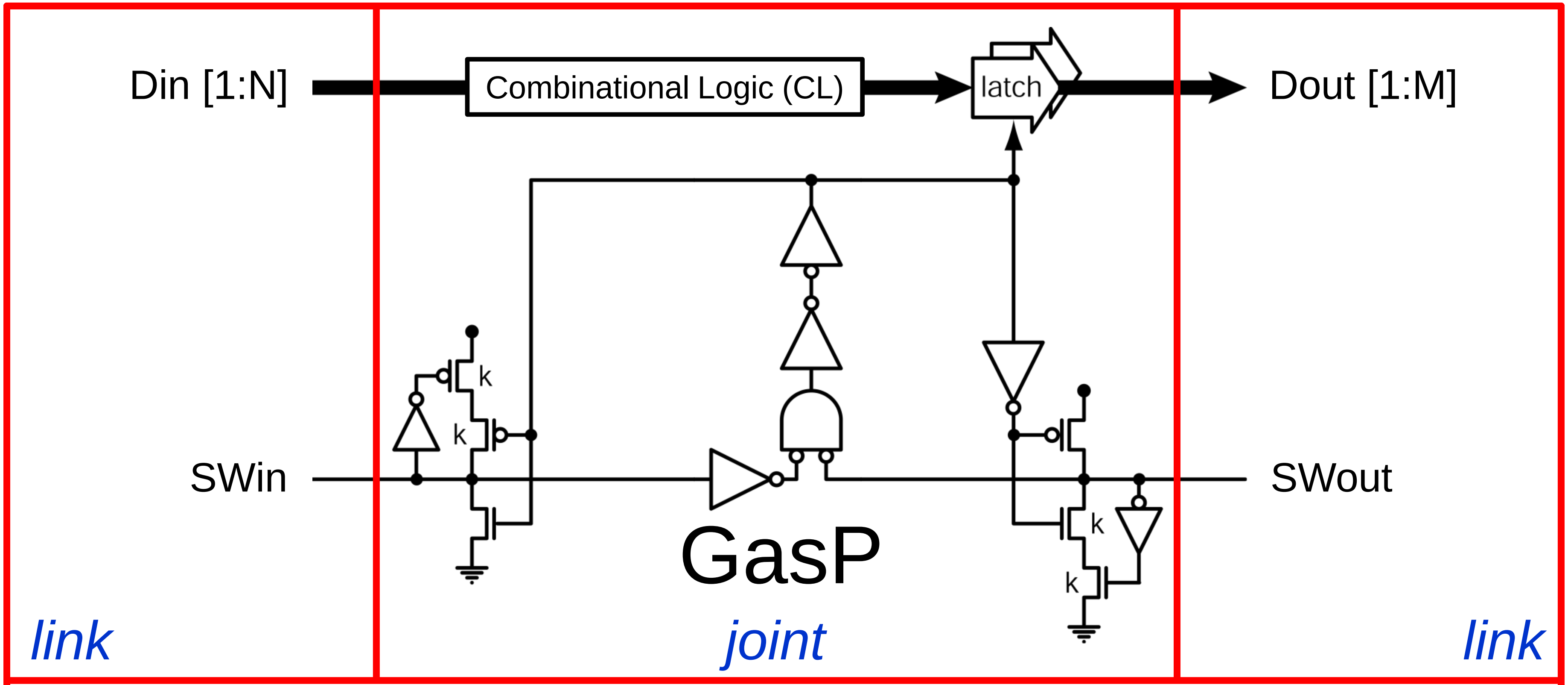
Dataflow pipeline: re-design



Solution:

distribute the weight

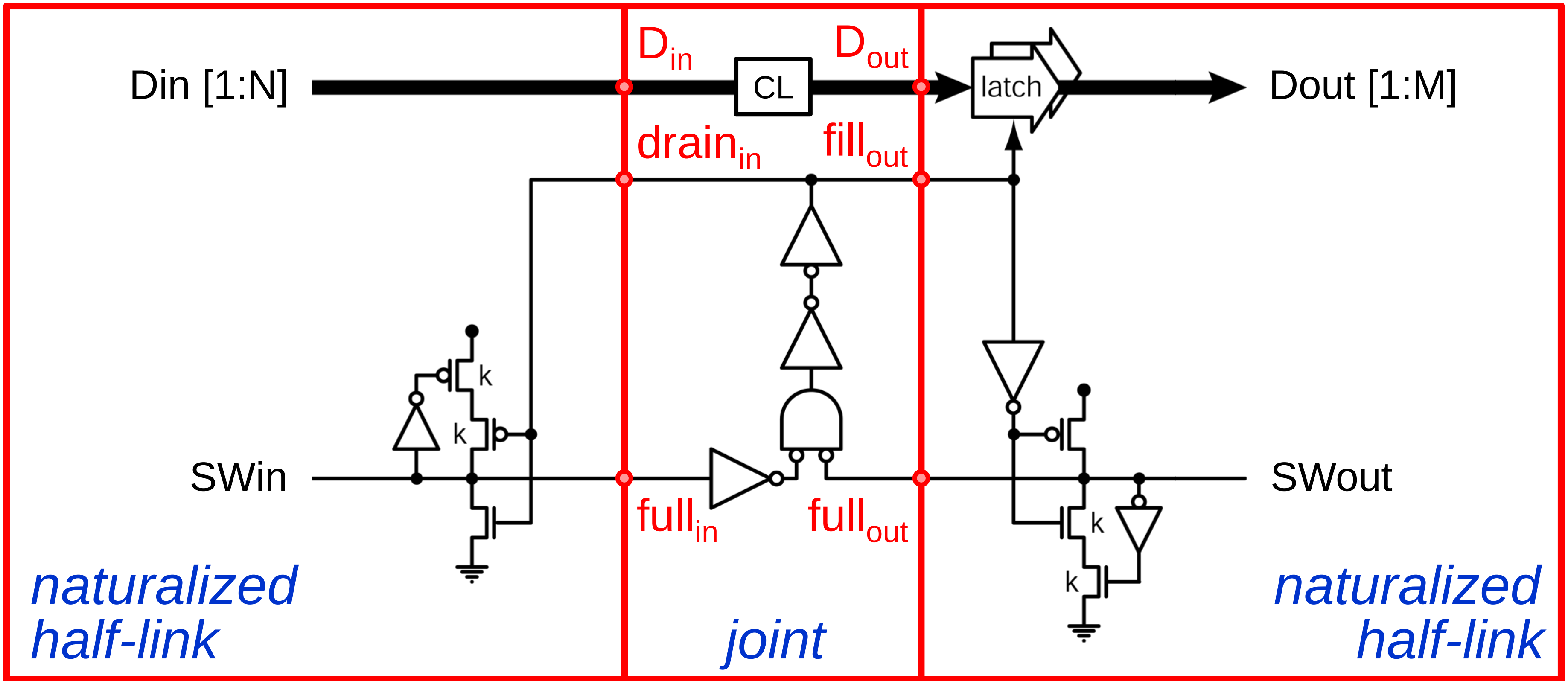
Dataflow pipeline: re-design from



Solution:

- move the link-joint interface
- by moving the communication logic from the joint to the links
- such that link-joint interface signals match those in the pipeline action

Dataflow pipeline: re-design to

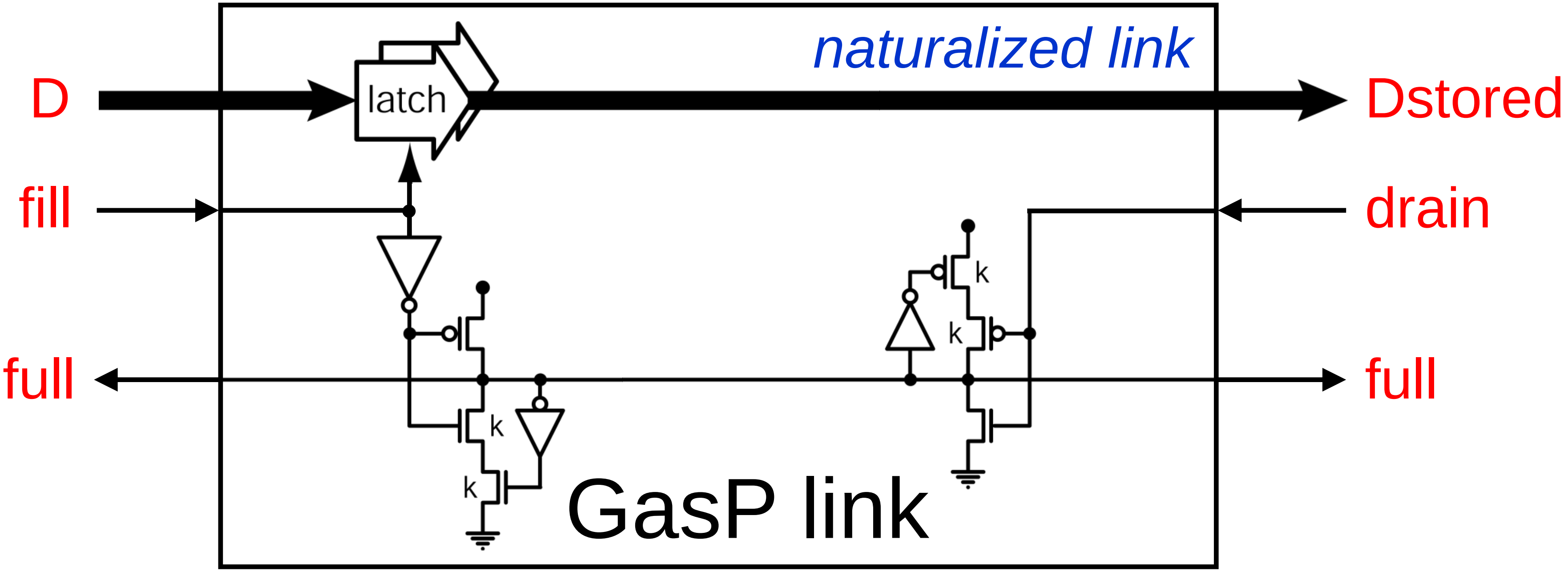


Solution:

- move the link-joint interface
- by moving the communication logic from the joint to the links
- such that link-joint interface signals match those in the pipeline action

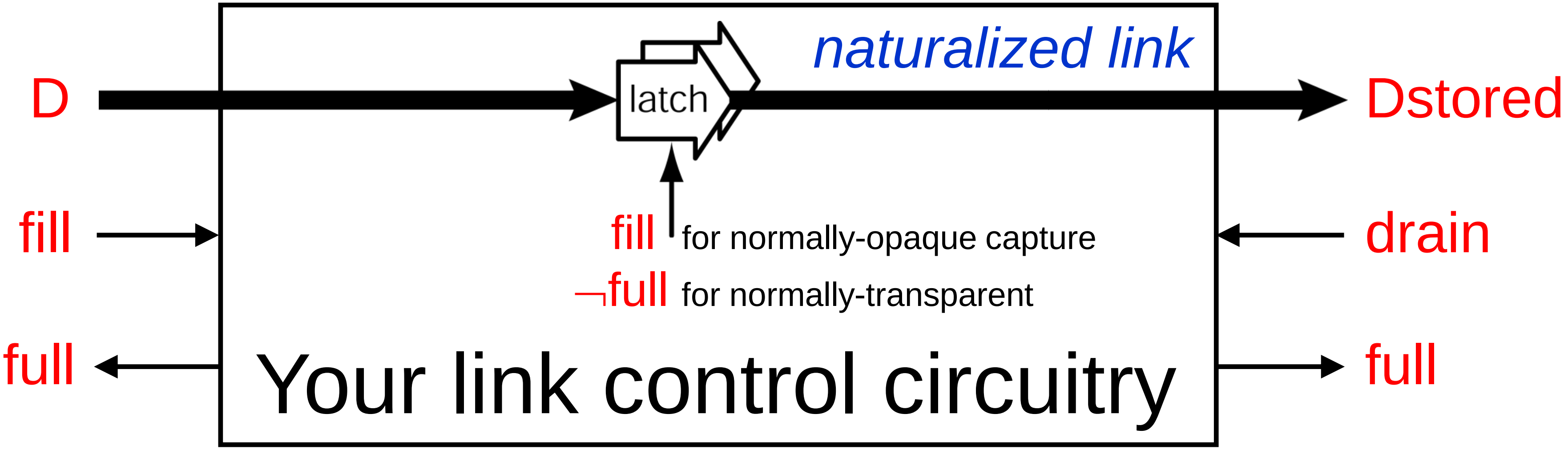
Naturalized communication: take-away

- by exposing the fundamental pipeline signals: full-empty, drain, fill, D
- we can standardize the link-joint interface
- and simplify + share designs and tools



Naturalized communication: take-away

- by exposing the fundamental pipeline signals: full-empty, drain, fill, D
- we can standardize the link-joint interface
- and simplify + share designs and tools

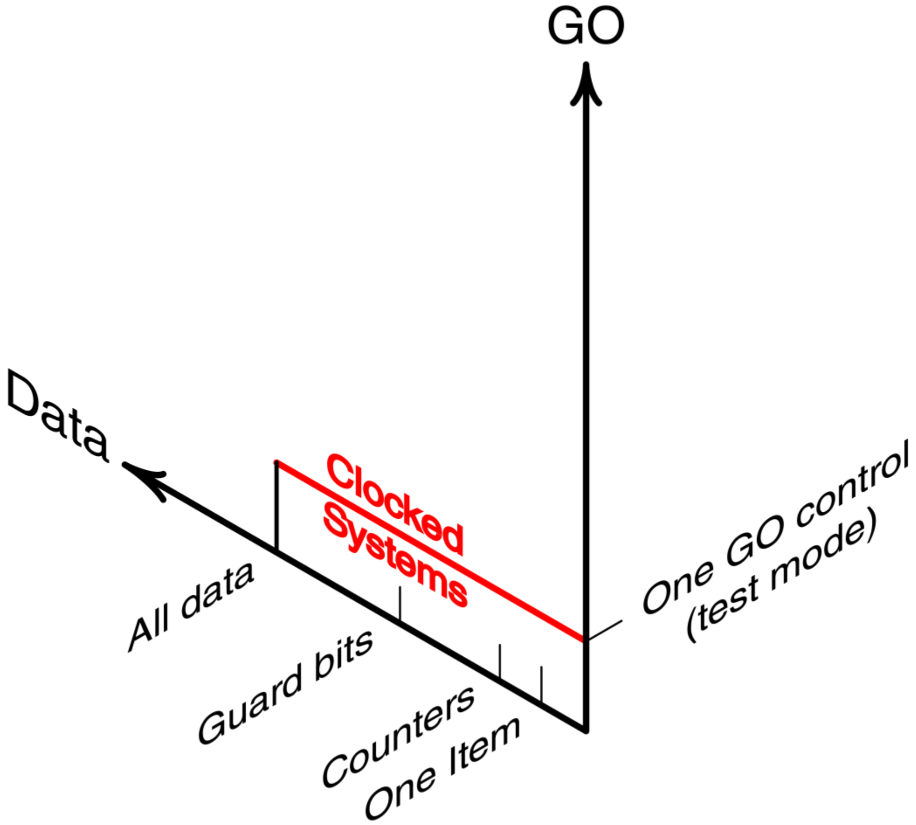


D
fill
full
latch
naturalized link
Dstored
drain
full
fill for normally-opaque capture
¬full for normally-transparent
Your link control circuitry

PART 2
naturalized testing
(silicon)

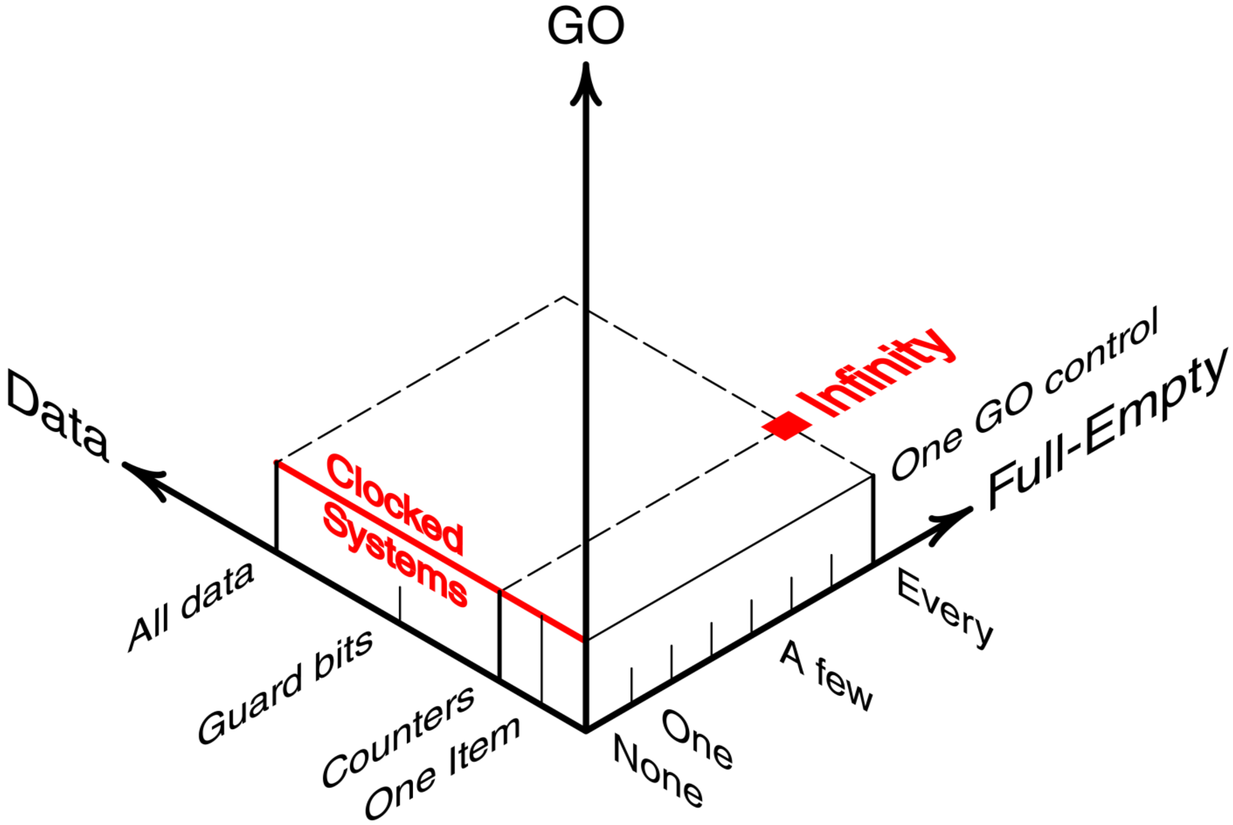
Naturalized testing: where we came from

- synchronous systems
- start and stop the global clock action : One GO control
- use scan test to control + observe global state : Data
- to detect stuck-at faults



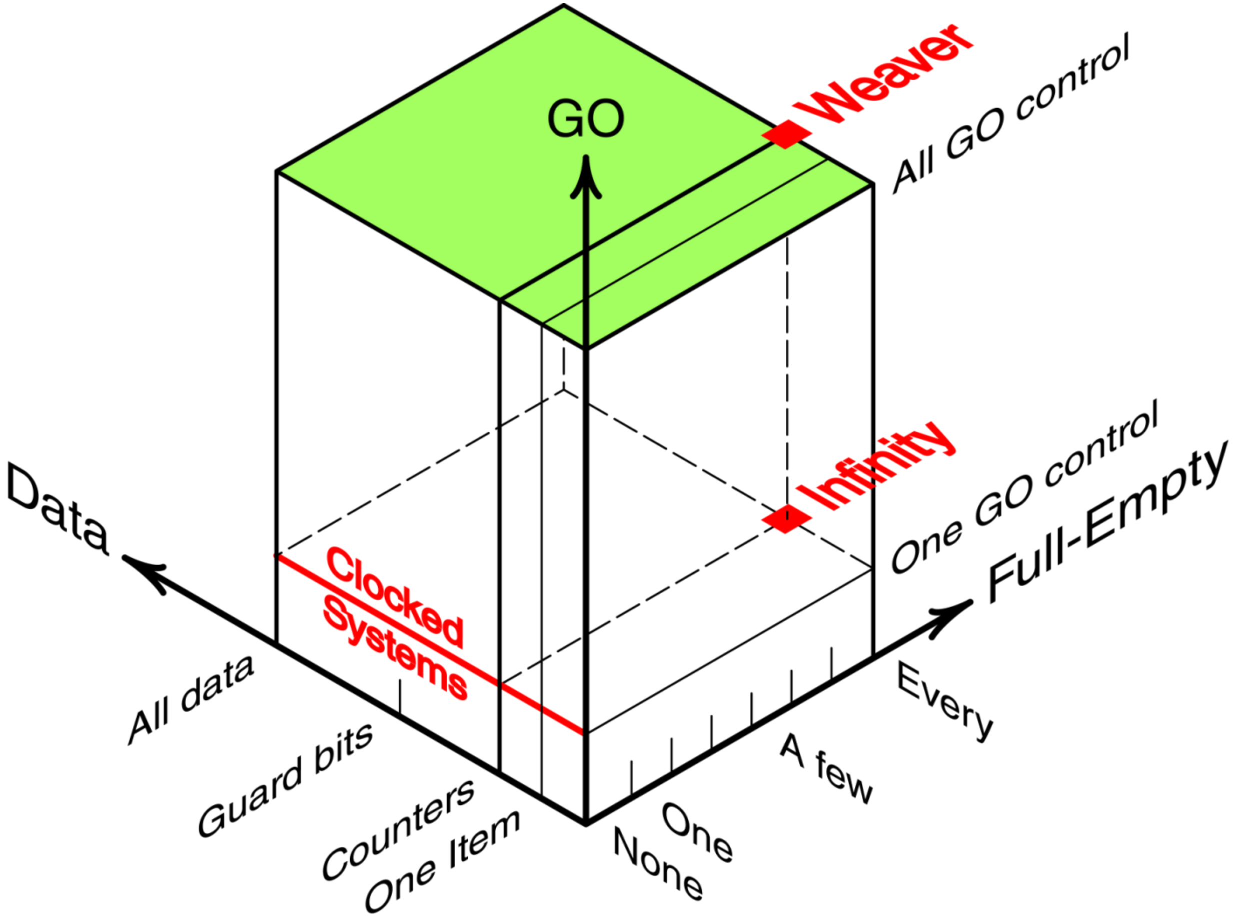
Naturalized testing: where we are

- self-timed systems
- start and stop all local actions together : One GO control
- use scan test to control + observe local state : Data + Full-Empty
- to detect stuck-at faults



Naturalized testing: and where we go

- stuck-at fault detection & beyond: at-speed test / debug / characterization
- start and stop each local action individually : All GO control

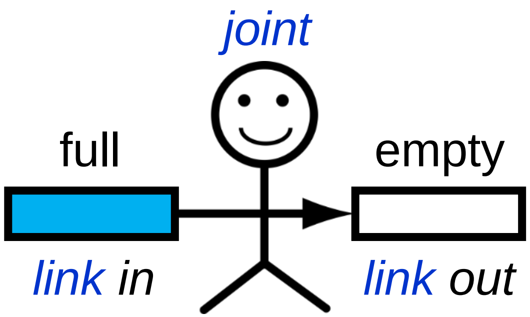


dedicated action control

Dataflow pipeline: action reminder

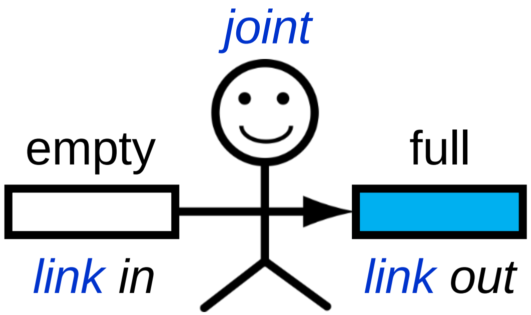
WHEN to act:

in is full
and
out is empty



WHAT to do:

- copy data
- drain *in*
- fill *out*



Dataflow pipeline: action with GO control

WHEN to act:

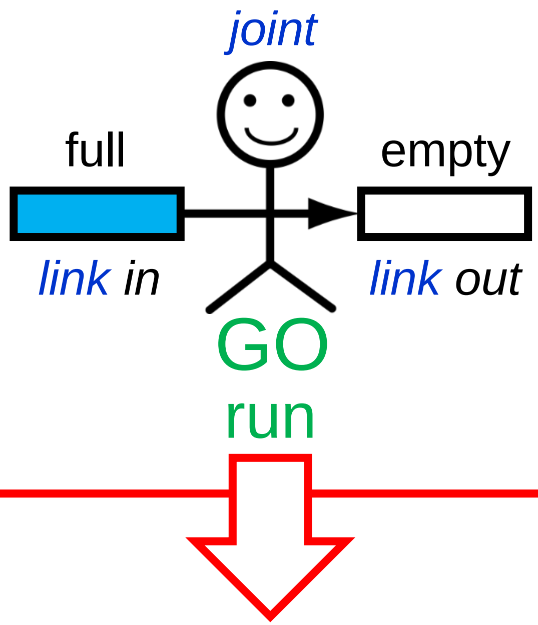
in is full

and

out is empty

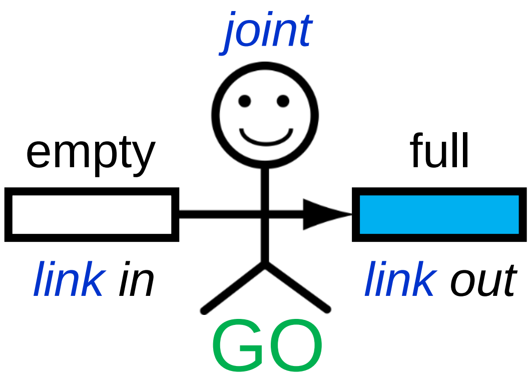
and

GO



WHAT to do:

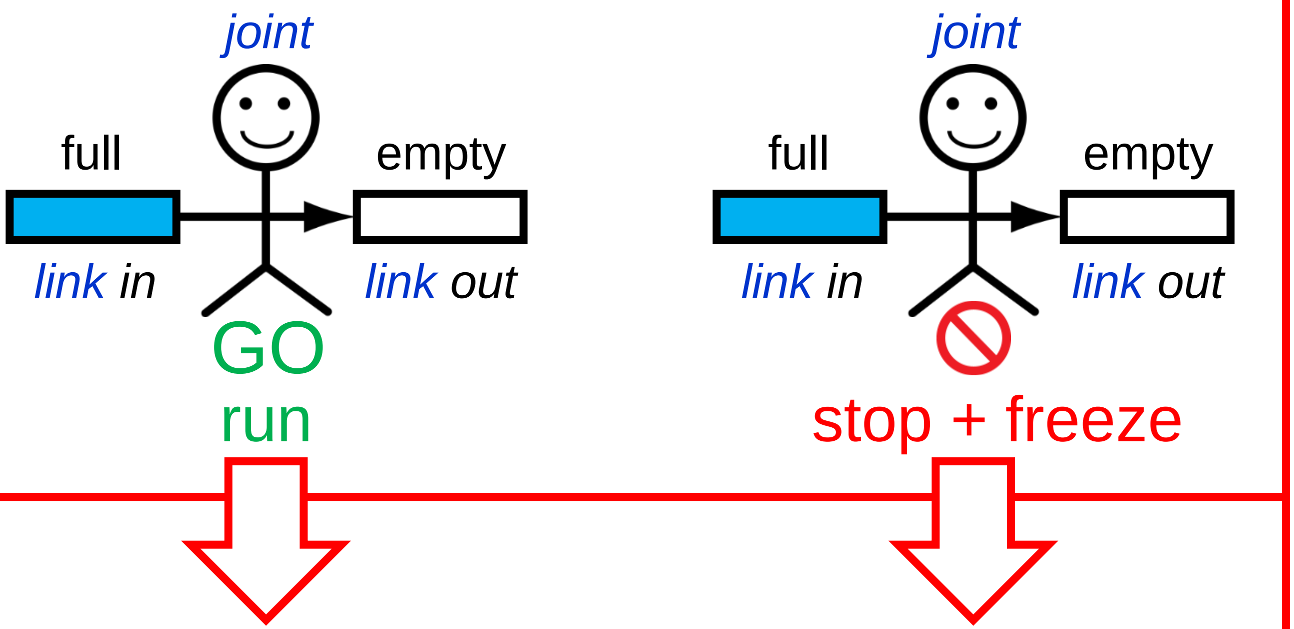
- copy data
- drain *in*
- fill *out*



Dataflow pipeline: action with GO control

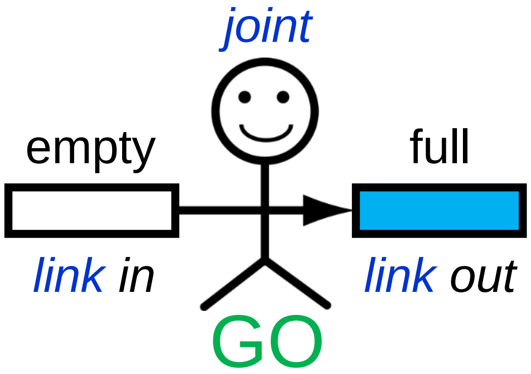
WHEN to act:

in is full
and
out is empty
and
GO



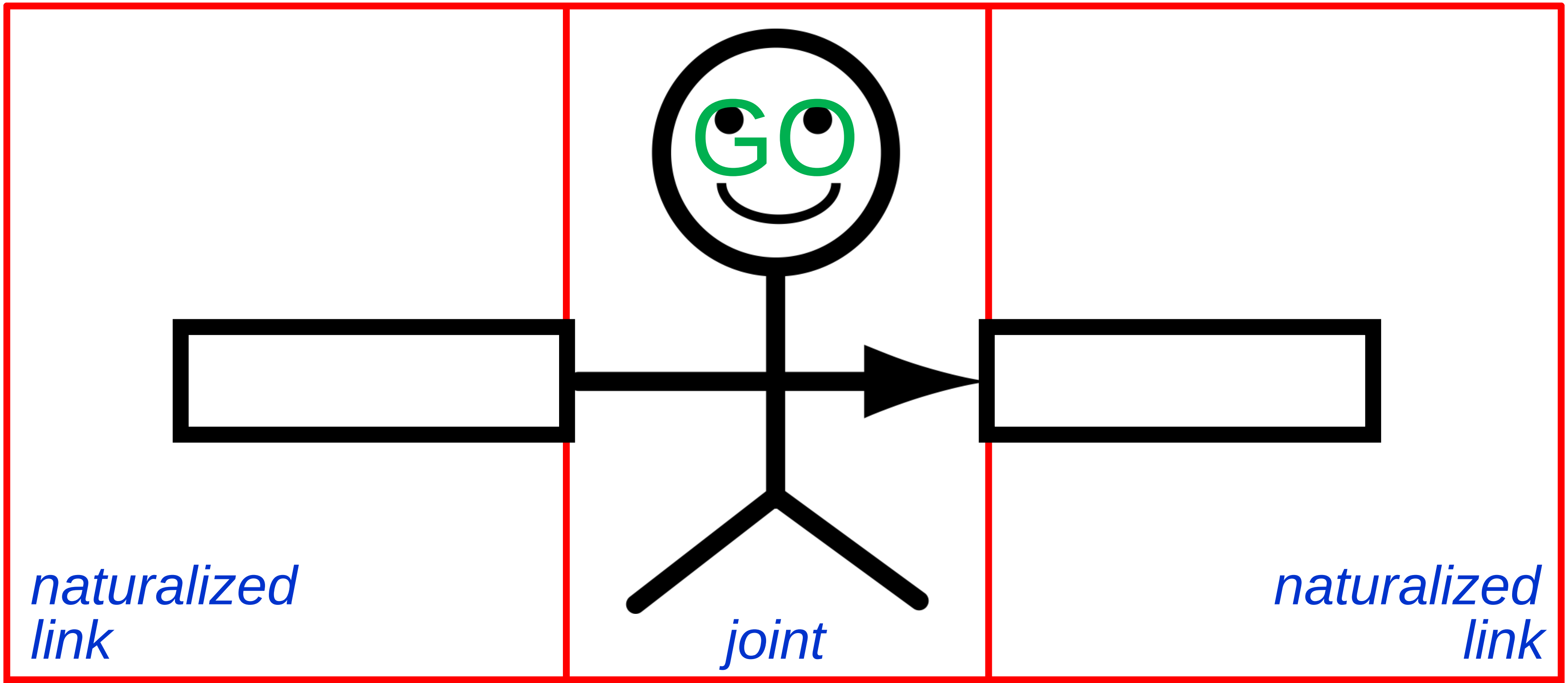
WHAT to do:

- copy data
- drain *in*
- fill *out*



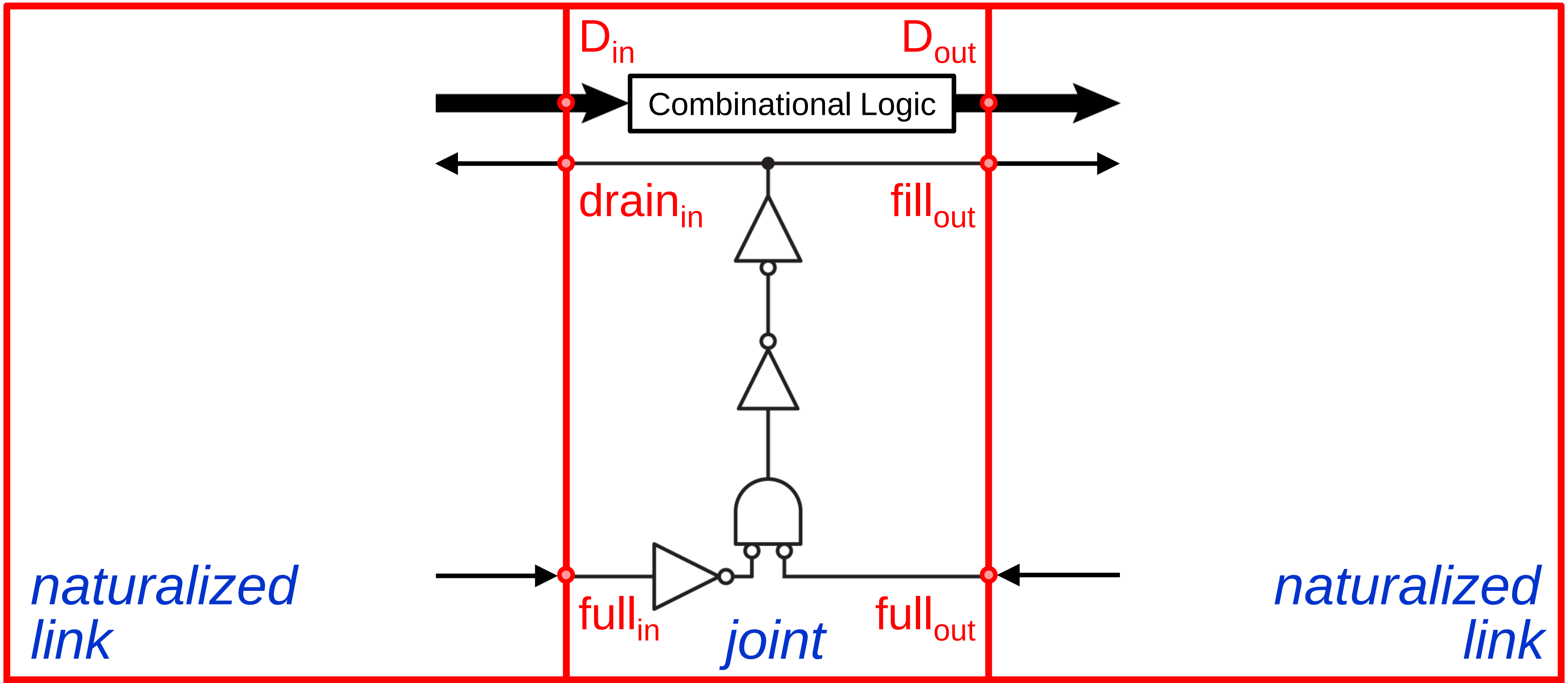
no action

Dataflow pipeline: design with GO control



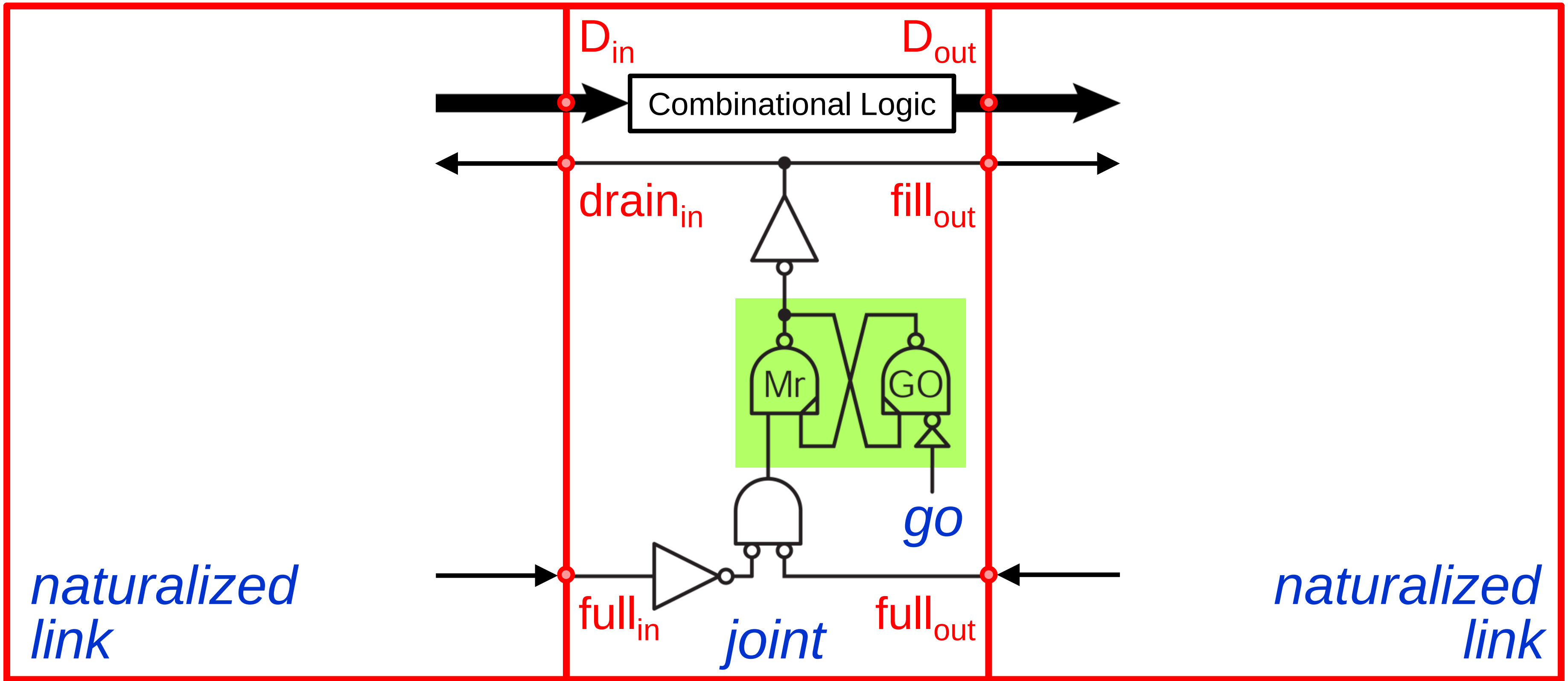
design reminder

Dataflow pipeline: design with GO control



design reminder

Dataflow pipeline: design with GO control



Solution MrGO:

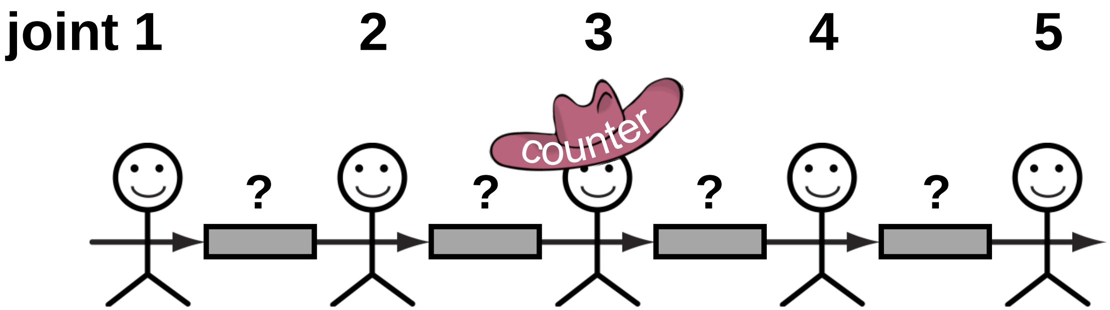
pronounced "Mister GO"

- *go* is high (GO) : run
- *go* is low (🚫) : stop and freeze
- arbiter for safe stop : "proper stopper"
- scan chain delivers *go* signals

AT-SPEED TESTING with MrGO
single data item

Testing a counter at speed

INITIALIZE



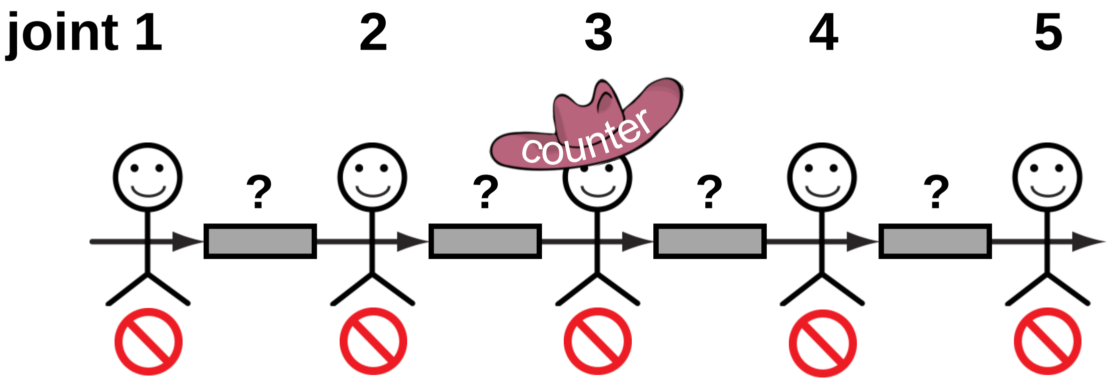
RUN

EVALUATE

Testing a counter at speed

INITIALIZE

1. freeze all joints



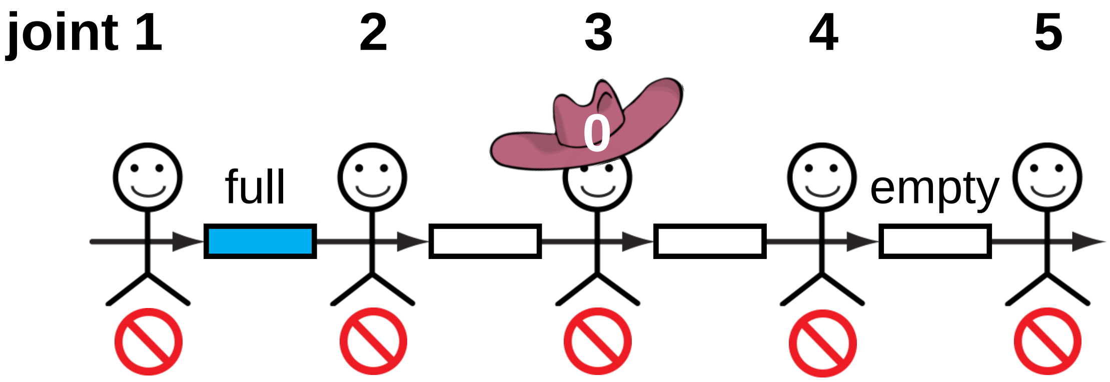
RUN

EVALUATE

Testing a counter at speed

INITIALIZE

1. freeze all joints
2. set state
    - full-empty links
    - counter data



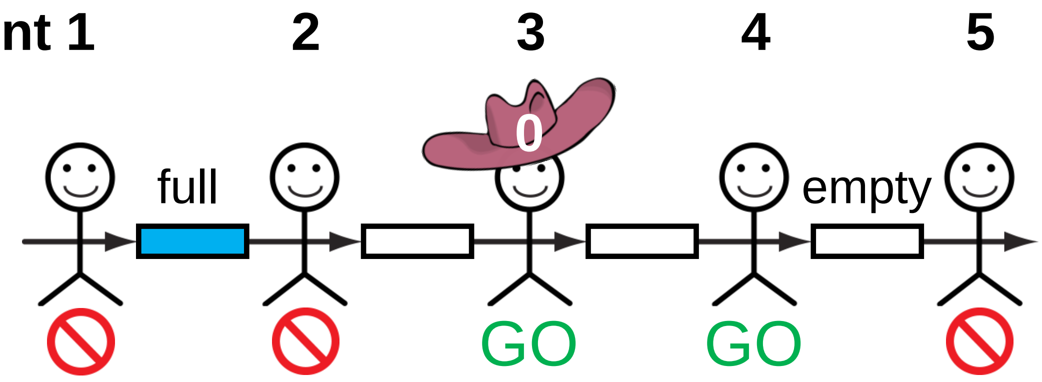
RUN

EVALUATE

Testing a counter at speed

INITIALIZE

1. freeze all joints
2. set state
   - full-empty links
   - counter data
3. unfreeze "runway" (3,4)



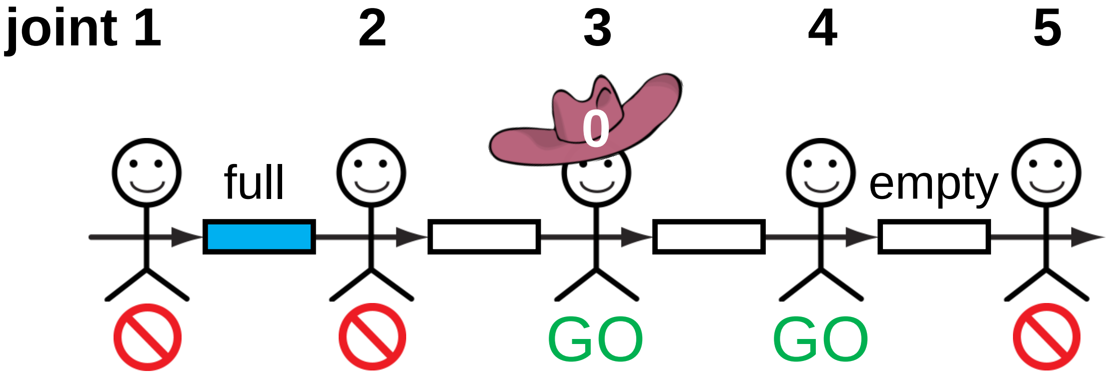
RUN

EVALUATE

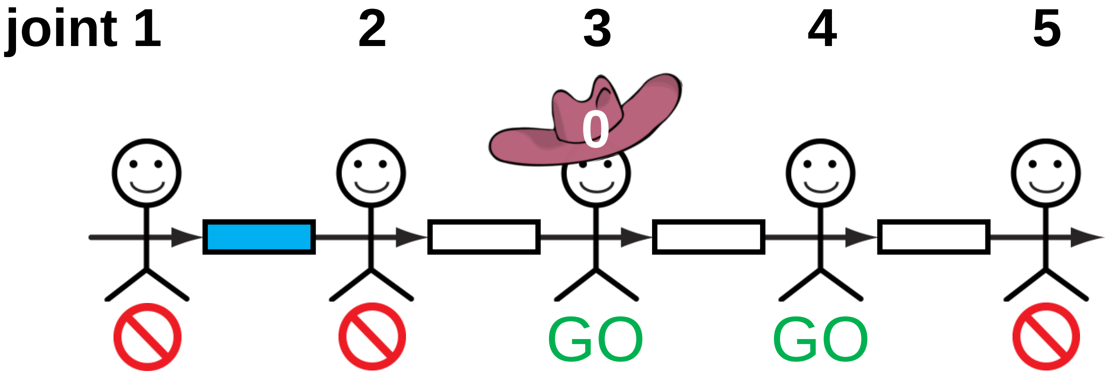
Testing a counter at speed

INITIALIZE

1. freeze all joints
2. set state
   - full-empty links
   - counter data
3. unfreeze "runway" (3,4)



RUN

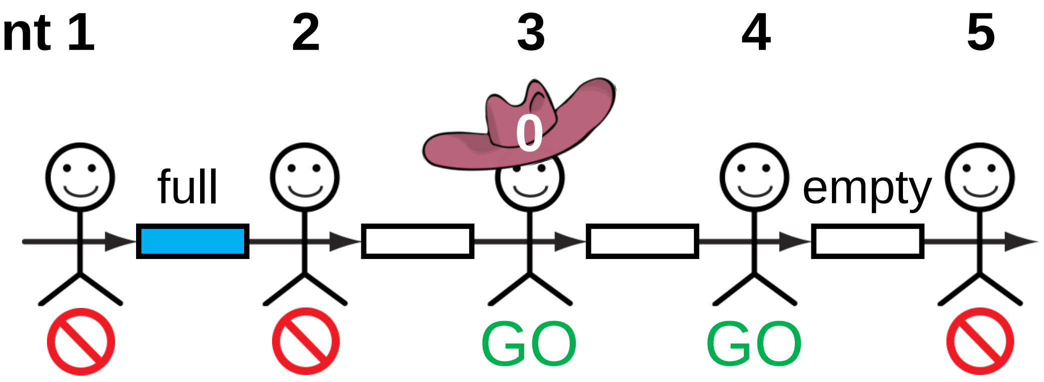


EVALUATE

Testing a counter at speed

INITIALIZE

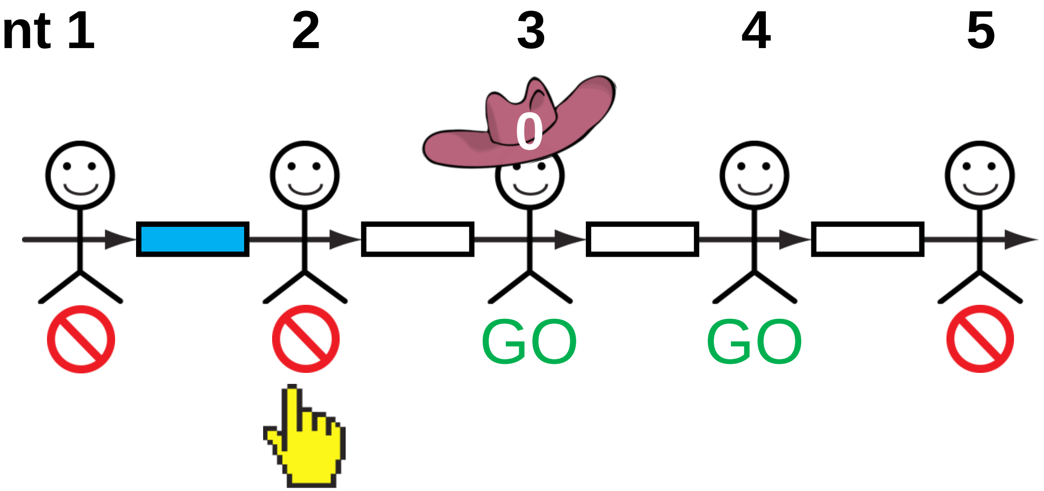
1. freeze all joints
2. set state
   - full-empty links
   - counter data
3. unfreeze "runway" (3,4)



RUN

1. unfreeze entry (2)
2. wait for action to finish

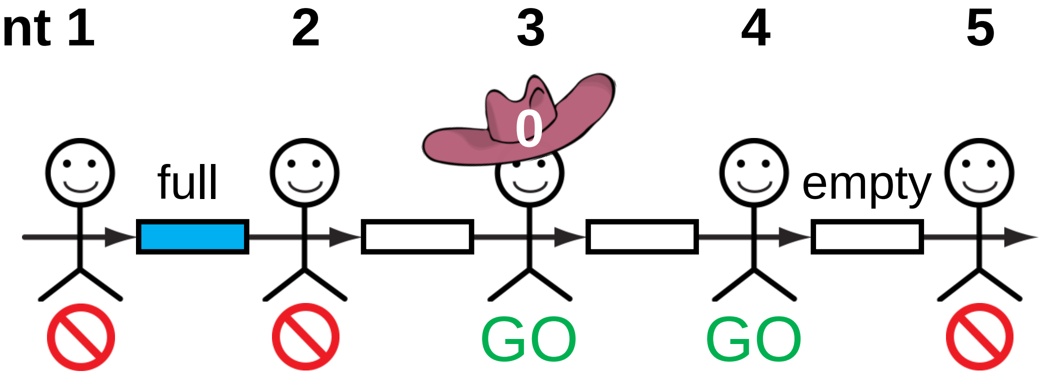
EVALUATE



Testing a counter at speed

INITIALIZE

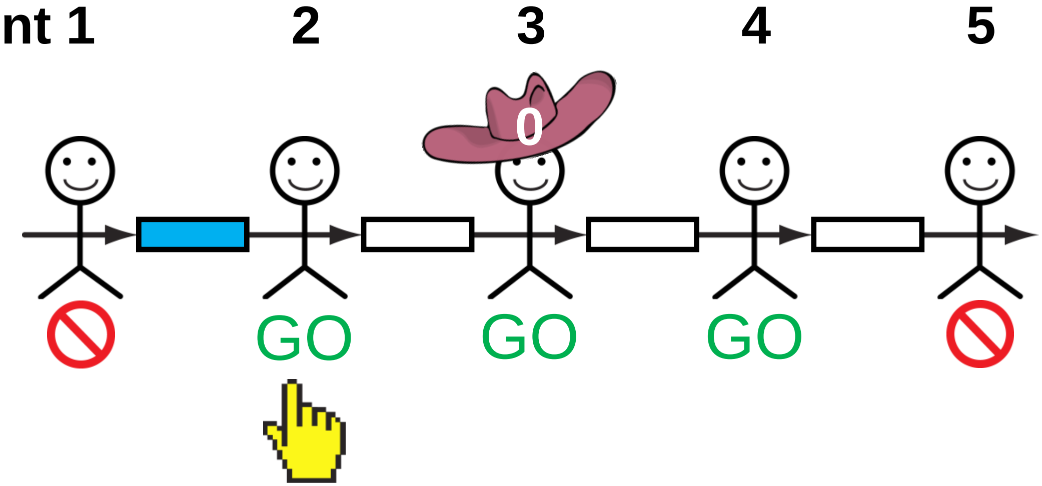
1. freeze all joints
2. set state
   - full-empty links
   - counter data
3. unfreeze "runway" (3,4)



RUN

1. unfreeze entry (2)
2. wait for action to finish

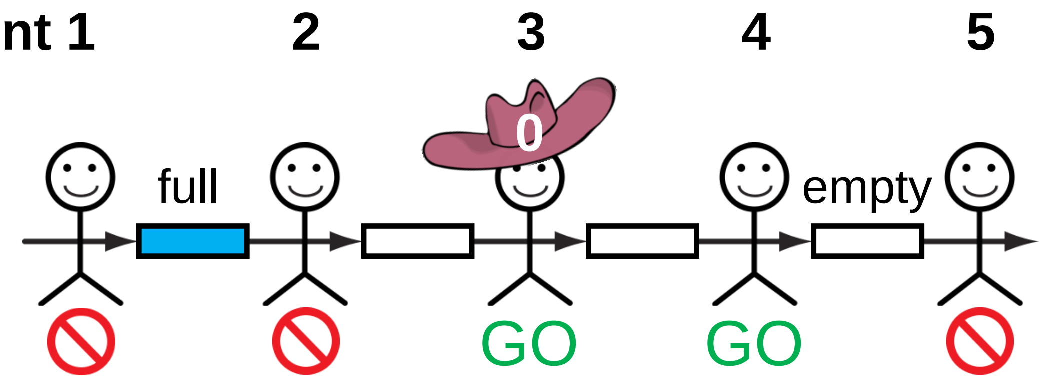
EVALUATE



Testing a counter at speed

INITIALIZE

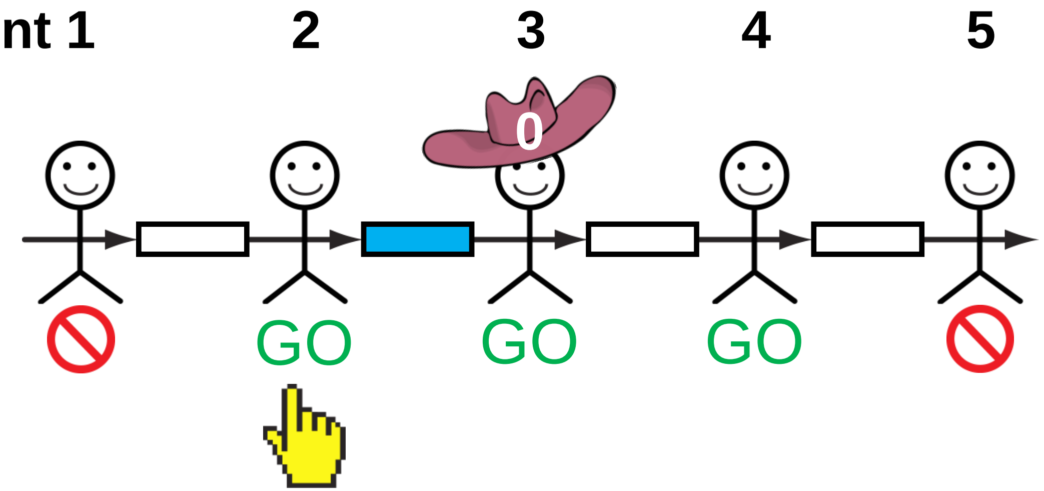
1. freeze all joints
2. set state
   - full-empty links
   - counter data
3. unfreeze "runway" (3,4)



RUN

1. unfreeze entry (2)
2. wait for action to finish

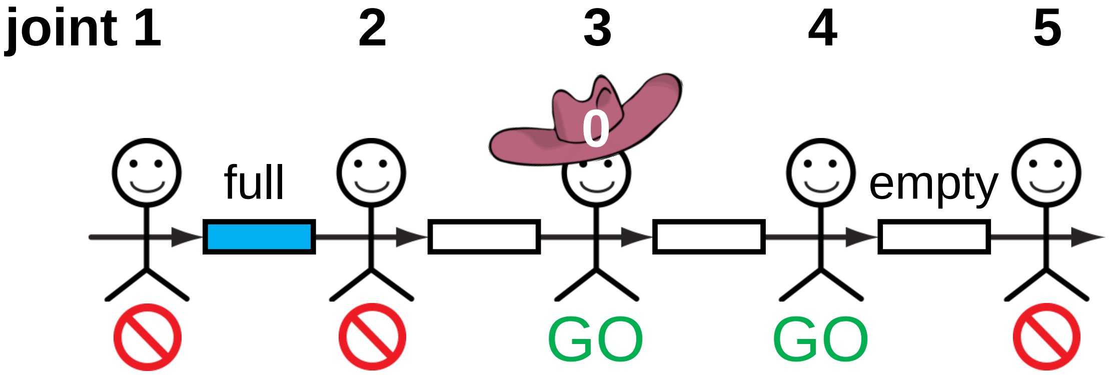
EVALUATE



Testing a counter at speed

INITIALIZE

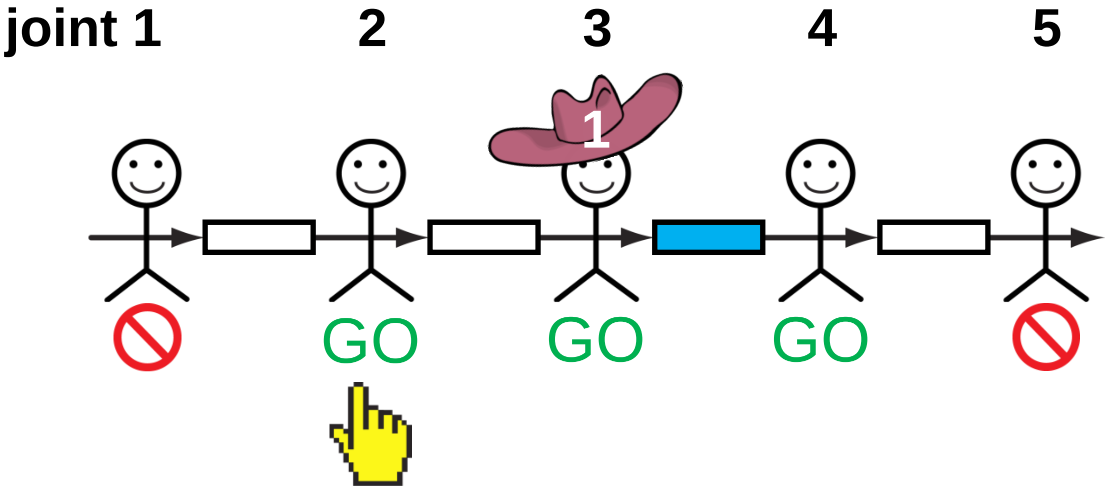
1. freeze all joints
2. set state
   - full-empty links
   - counter data
3. unfreeze "runway" (3,4)



RUN

1. unfreeze entry (2)
2. wait for action to finish

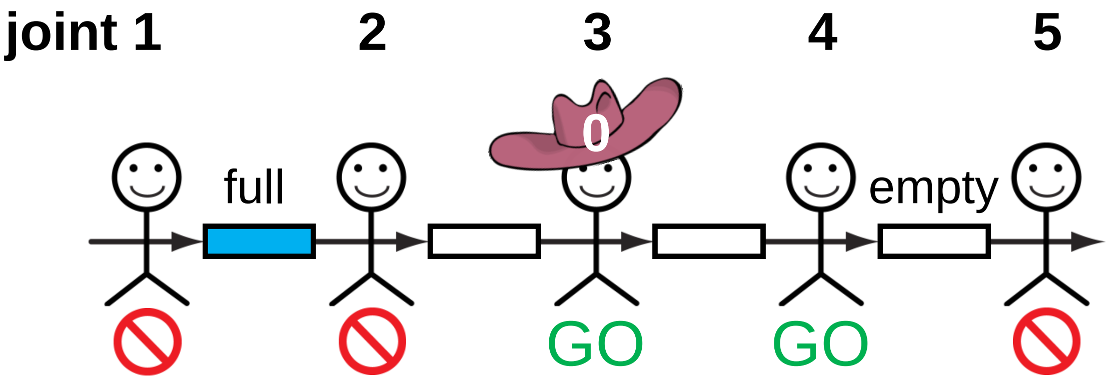
EVALUATE



Testing a counter at speed

INITIALIZE

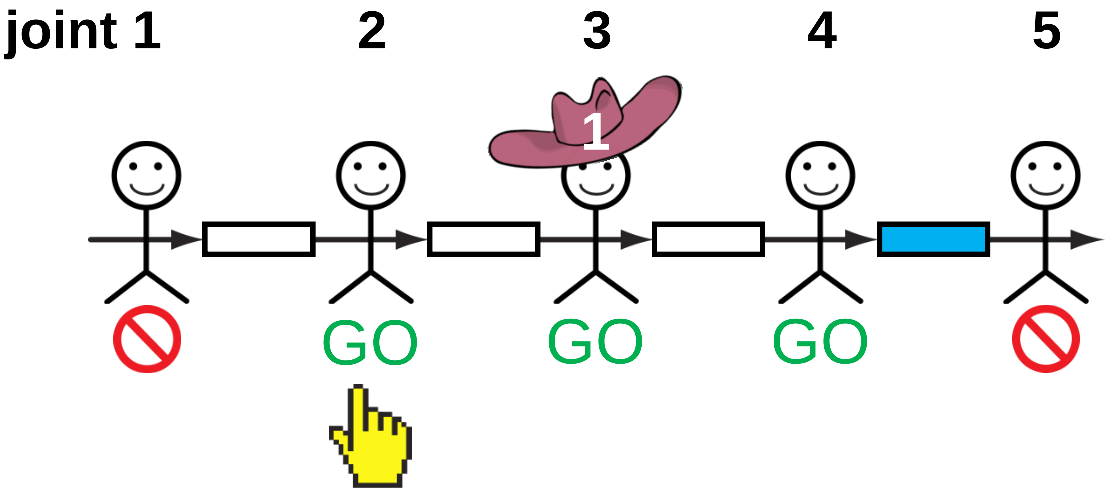
1. freeze all joints
2. set state
    - full-empty links
    - counter data
3. unfreeze "runway" (3,4)



RUN

1. unfreeze entry (2)
2. wait for action to finish

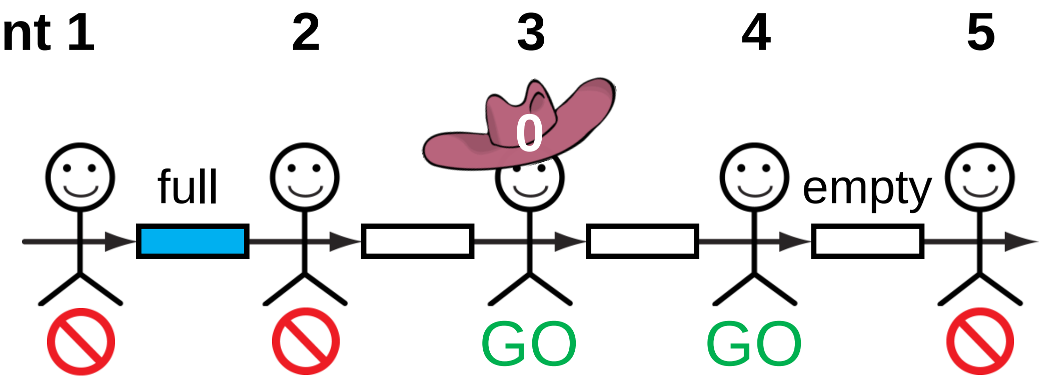
EVALUATE



Testing a counter at speed

INITIALIZE

1. freeze all joints
2. set state
   - full-empty links
   - counter data
3. unfreeze "runway" (3,4)

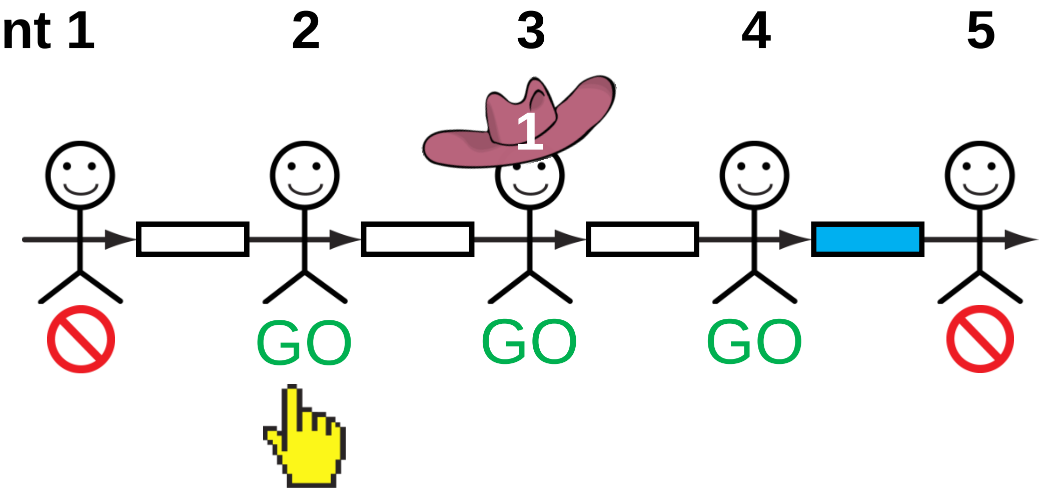


RUN

1. unfreeze entry (2)
2. wait for action to finish

EVALUATE

- read counter data

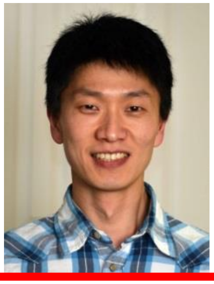


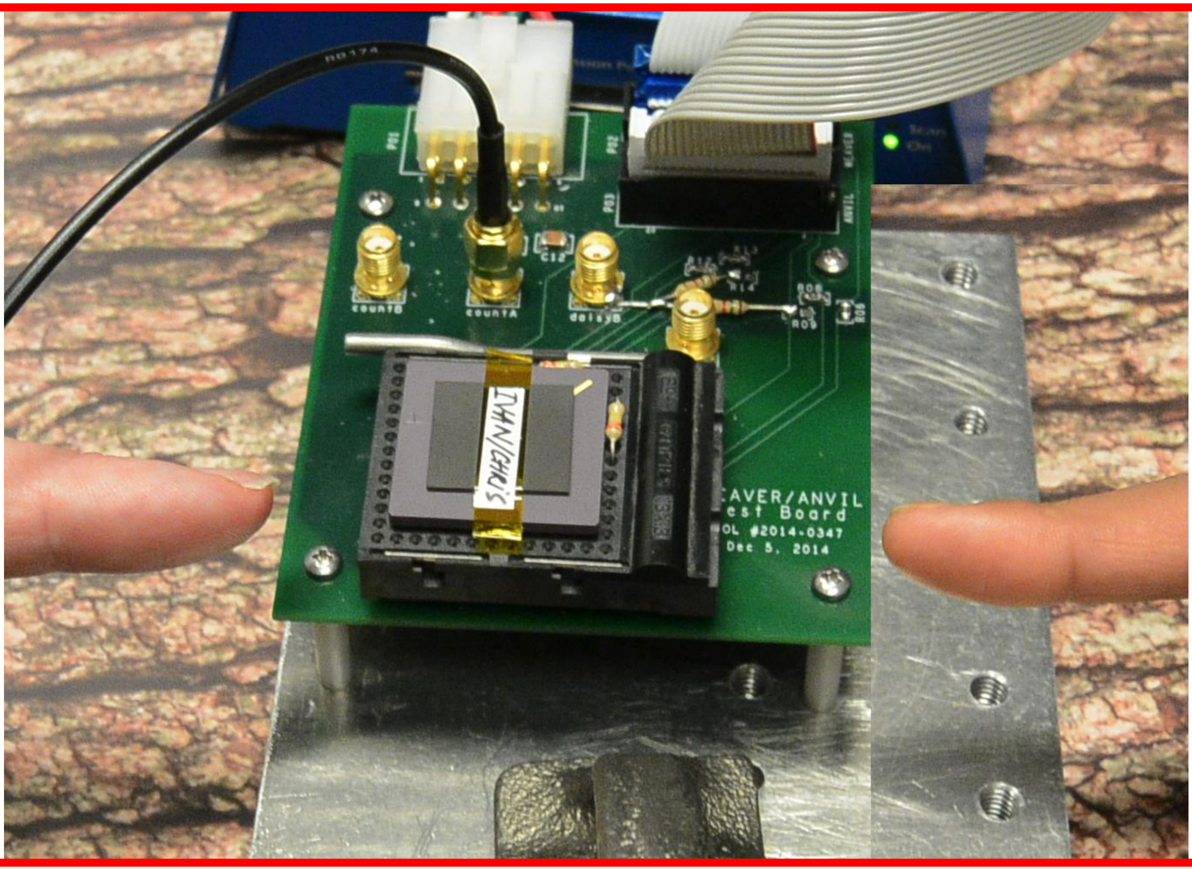
Get real!

- two working silicon experiments – Weaver and Anvil
- use MrGO + JTAG-scan-access for test, debug, and characterization
- LIVE demos and tests are available at the conference

marly

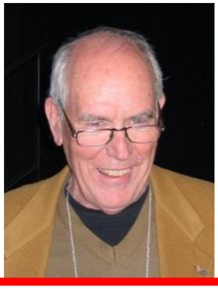
swetha


hoon



nav

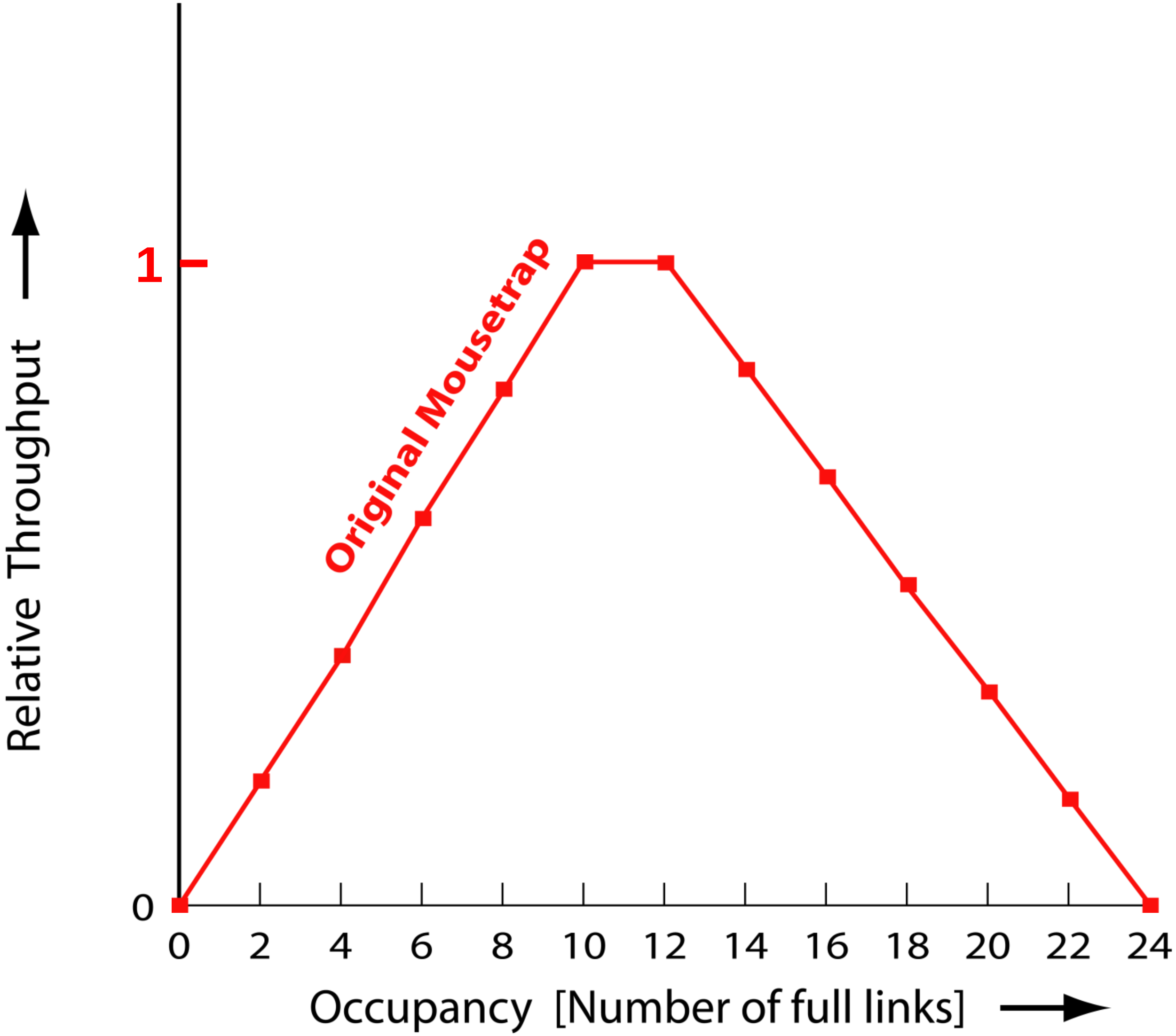
chris


ivan

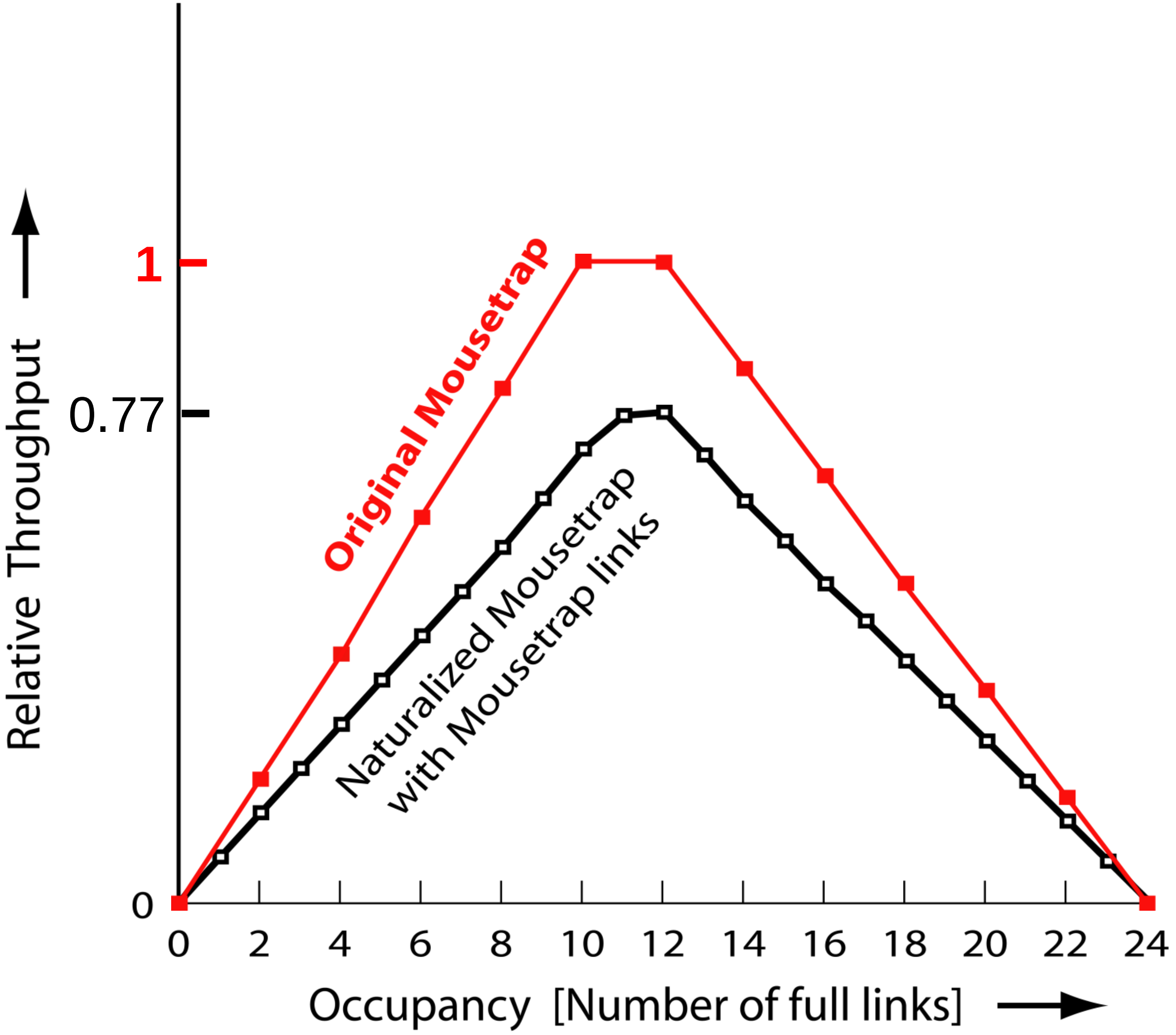
BACK-UP SLIDES

THROUGHPUT
original and naturalized Mousetrap

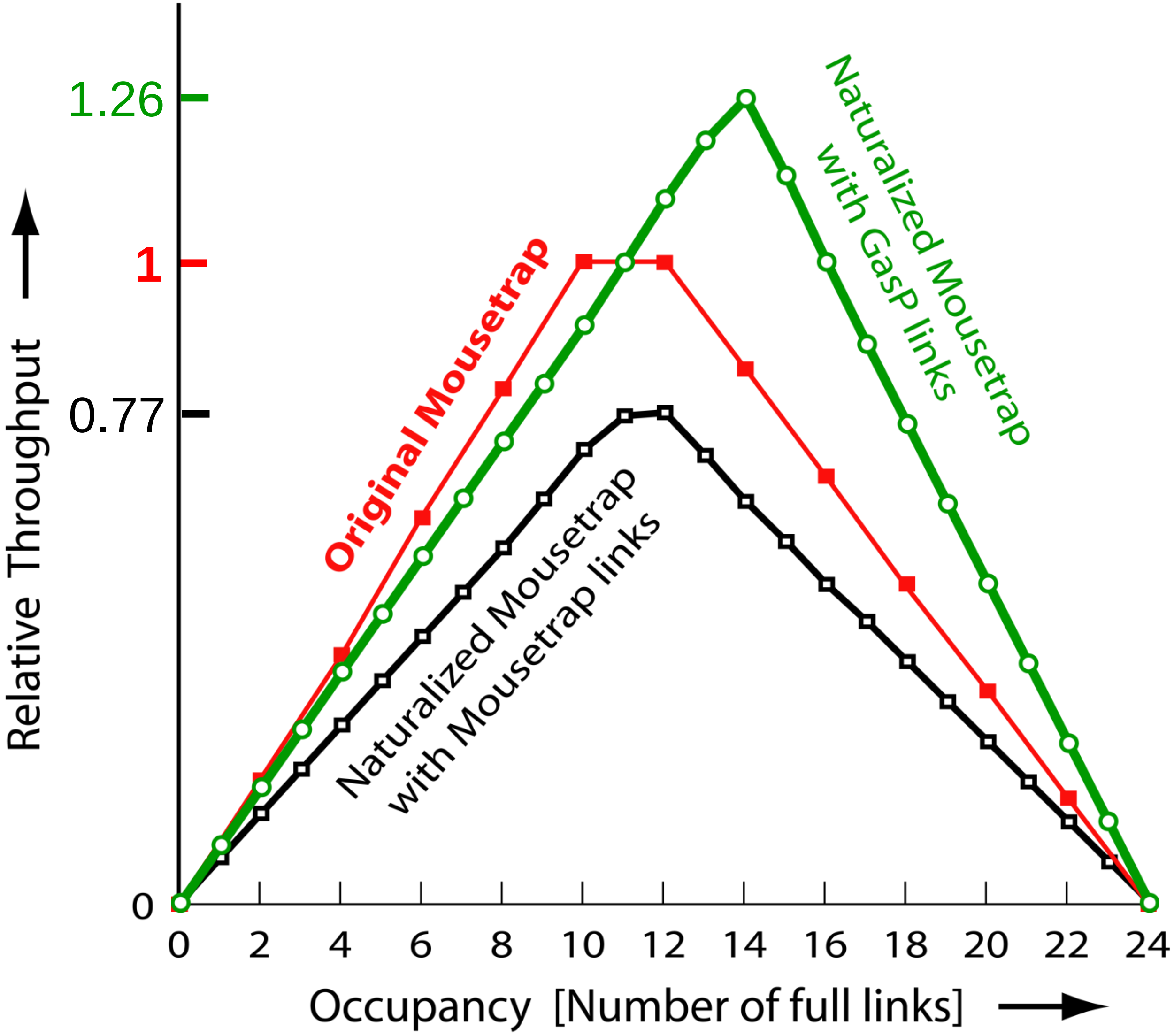
Throughput comparison: canopy graphs



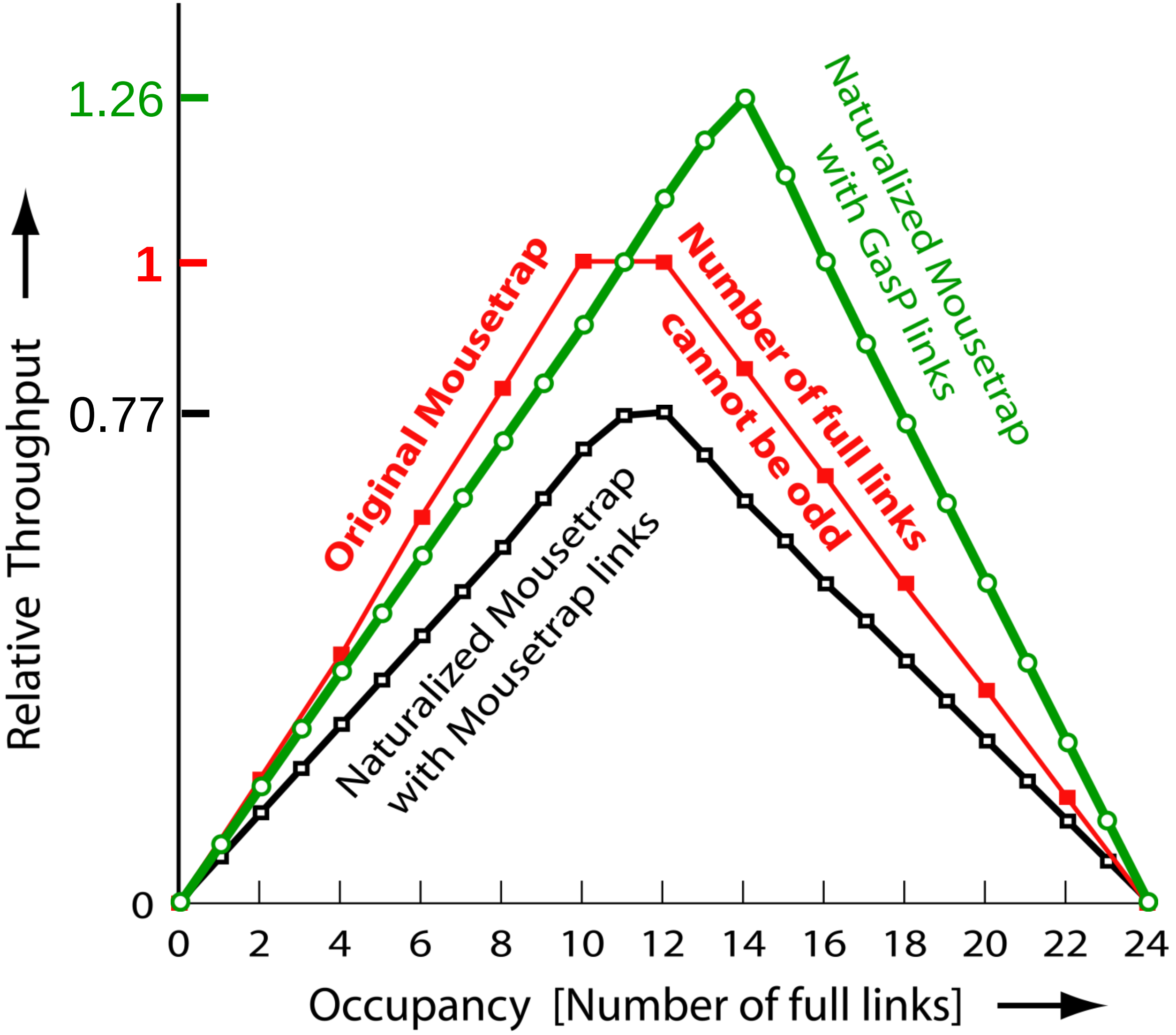
Throughput comparison: canopy graphs

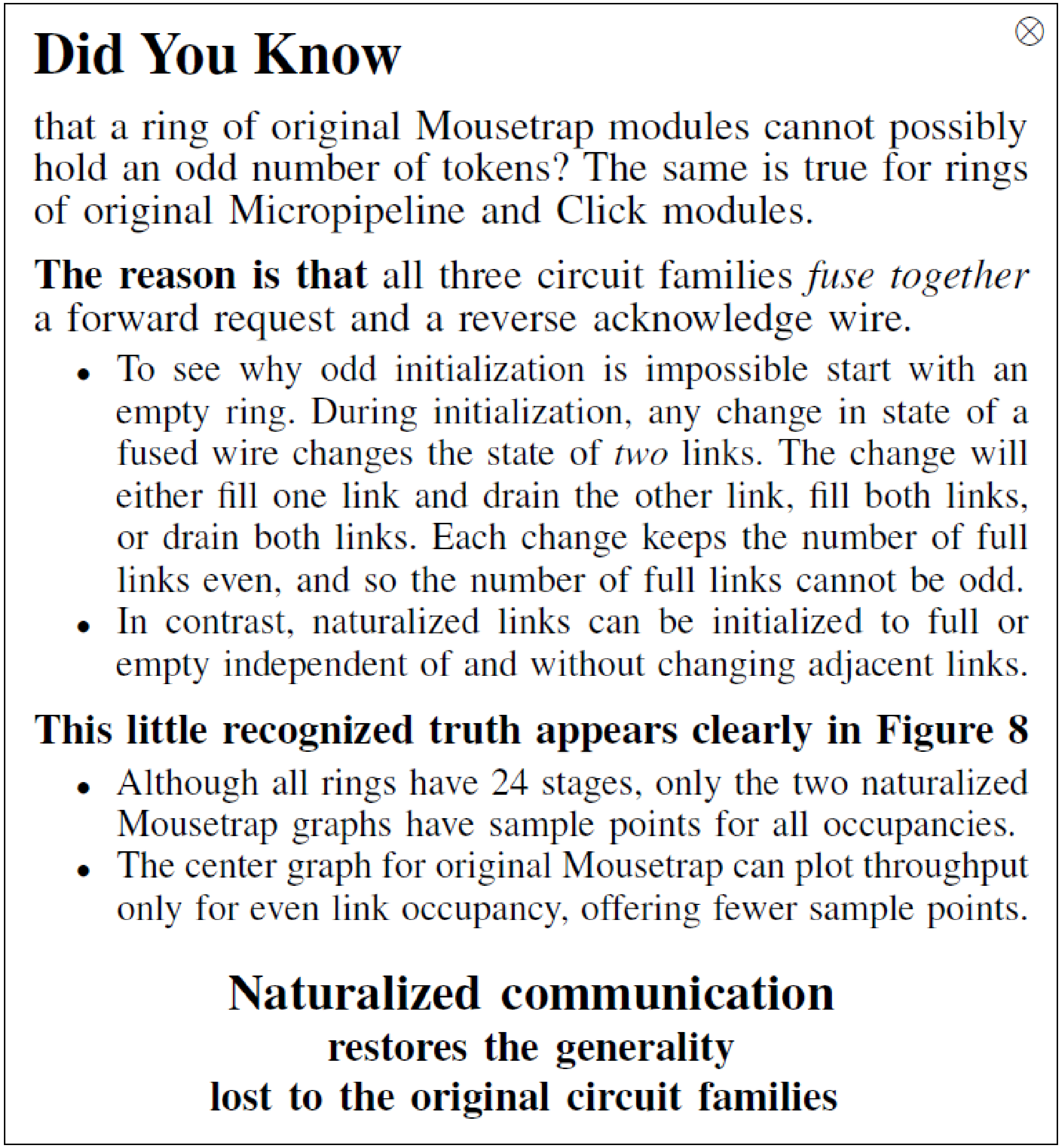


Throughput comparison: canopy graphs



Throughput comparison: canopy graphs





Did You Know

that a ring of original Mousetrap modules cannot possibly hold an odd number of tokens? The same is true for rings of original Micropipeline and Click modules.

The reason is that all three circuit families *fuse together* a forward request and a reverse acknowledge wire.

- To see why odd initialization is impossible start with an empty ring. During initialization, any change in state of a fused wire changes the state of *two* links. The change will either fill one link and drain the other link, fill both links, or drain both links. Each change keeps the number of full links even, and so the number of full links cannot be odd.
- In contrast, naturalized links can be initialized to full or empty independent of and without changing adjacent links.

This little recognized truth appears clearly in Figure 8

- Although all rings have 24 stages, only the two naturalized Mousetrap graphs have sample points for all occupancies.
- The center graph for original Mousetrap can plot throughput only for even link occupancy, offering fewer sample points.

Naturalized communication
restores the generality
lost to the original circuit families

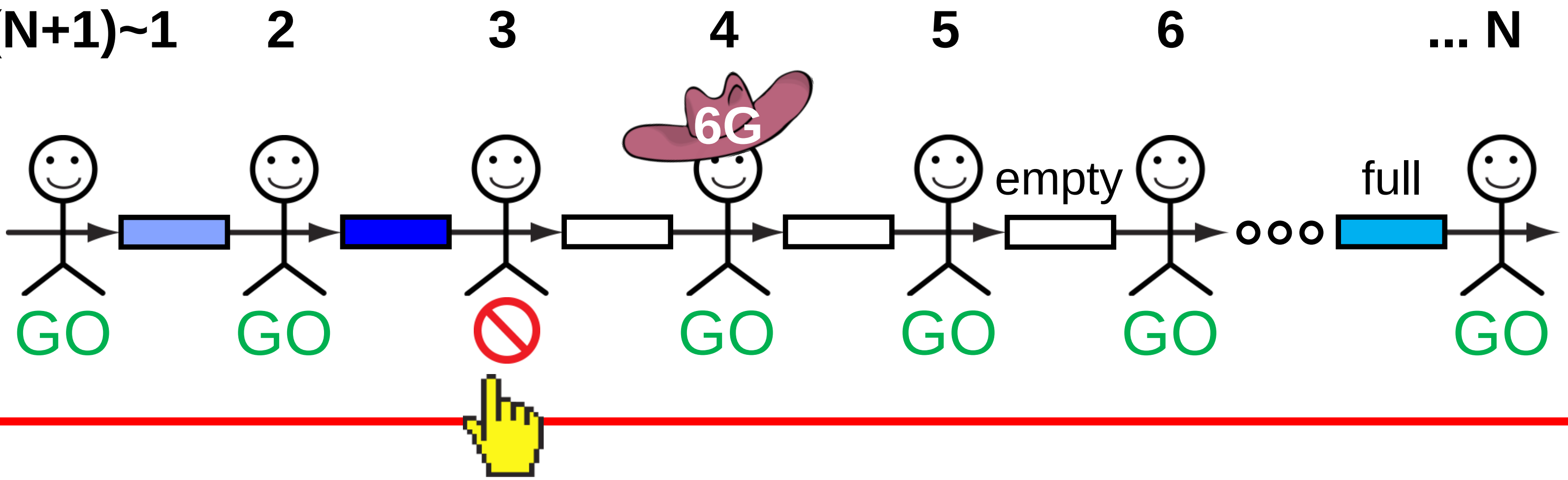
DELETE-button
added especially
for Jens Sparsø

CANOPY GRAPHS
characterization with MrGO

Creating canopy graphs

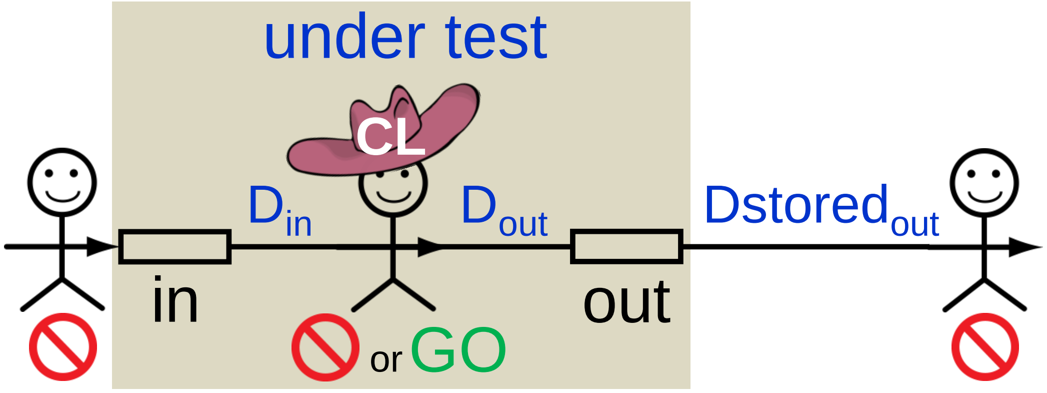
```
DO (ALL > i > 0 links)
   counter=0
   run 1 second with i full links
   arbitrated stop
   read counter
OD
```

FINAL for *i* ~ 60% links



STUCK-AT FAULTS
one-shot testing with MrGO

Testing stuck-at faults



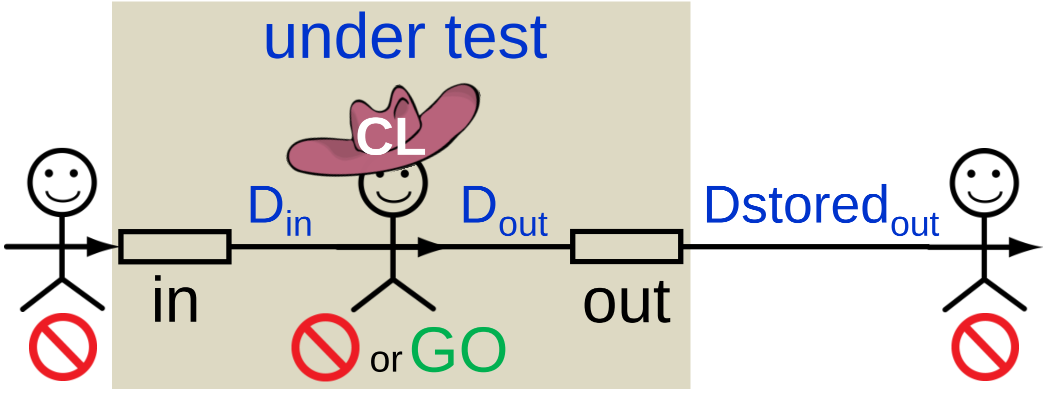
TEST control logic

```
DO (ALL full-empty link combos)
   freeze joint
   set $full_{in}$  = combo(in)
         $full_{out}$ = combo(out)
   evaluate if links remain unchanged
   unfreeze joint
   evaluate final link states
OD
```

TEST datapath (normally opaque)

```
DO (ALL CL test inputs)
   freeze joint
   set $full_{in}$        = TRUE
         $full_{out}$       = FALSE
         $D_{in}$           = test input
         $Dstored_{out}$ = $\neg CL(D_{in})$
   evaluate if $Dstored_{out}$ remain unchanged
   unfreeze joint
   evaluate if $Dstored_{out}$ = $CL(D_{in})$
OD
```

Testing stuck-at faults



TEST control logic

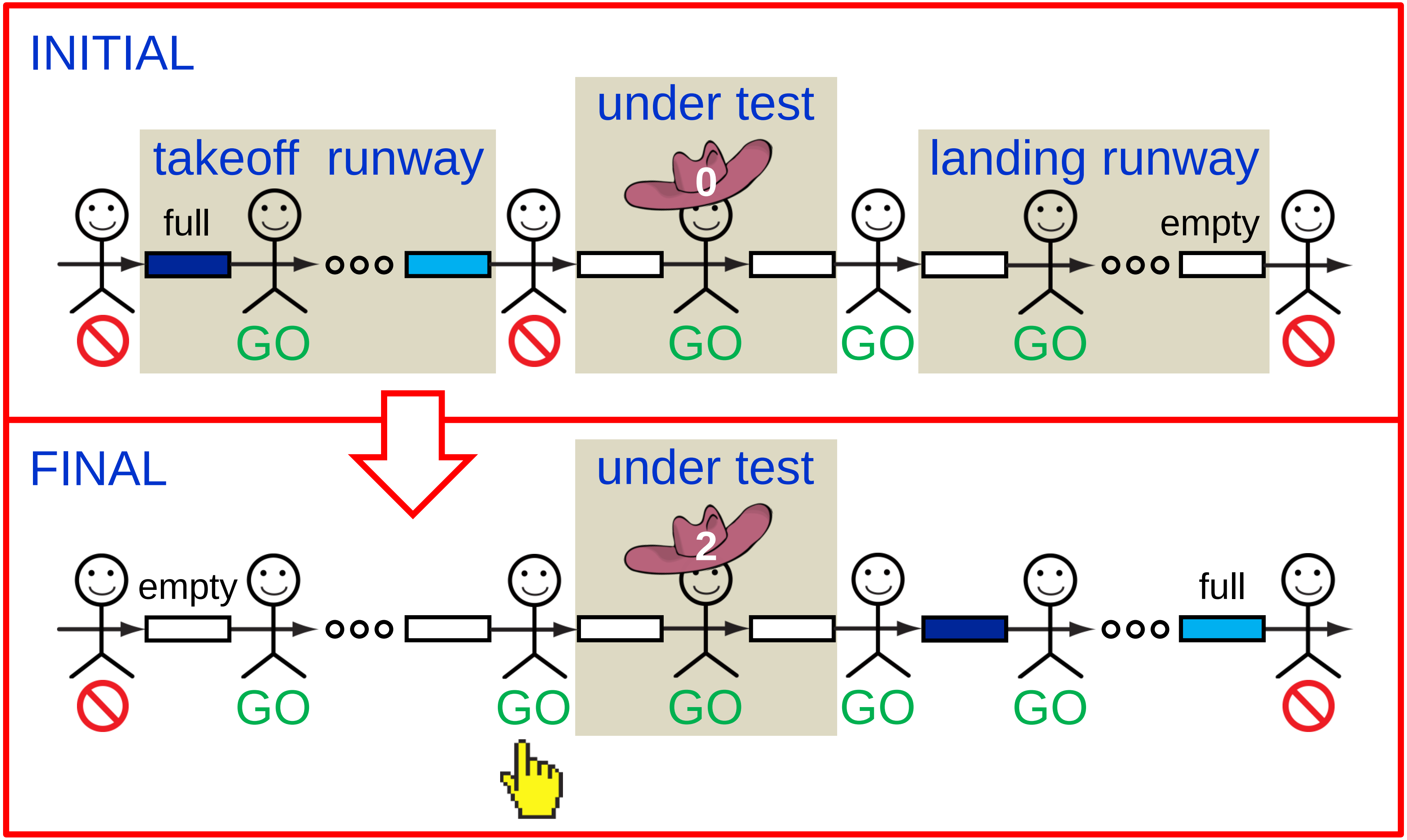
```
DO (ALL full-empty link combos)
   freeze joint
   set $full_{in}$ = combo(in)
         $full_{out}$ = combo(out)
   evaluate if links remain unchanged
   unfreeze joint
   evaluate final link states
OD
```

TEST datapath (normally transparent)

```
DO (ALL CL test inputs)
   freeze joint
   set $full_{in}=full_{out}$ = TRUE
         $D_{in}$ = test input
         $Dstored_{out}$ = $\neg CL(D_{in})$
   evaluate if $Dstored_{out}$ remain unchanged
   set $full_{out}$ = FALSE
   unfreeze joint
   evaluate if $Dstored_{out}$ = $CL(D_{in})$
OD
```

AT-SPEED TESTING
# of data burst with MrGO

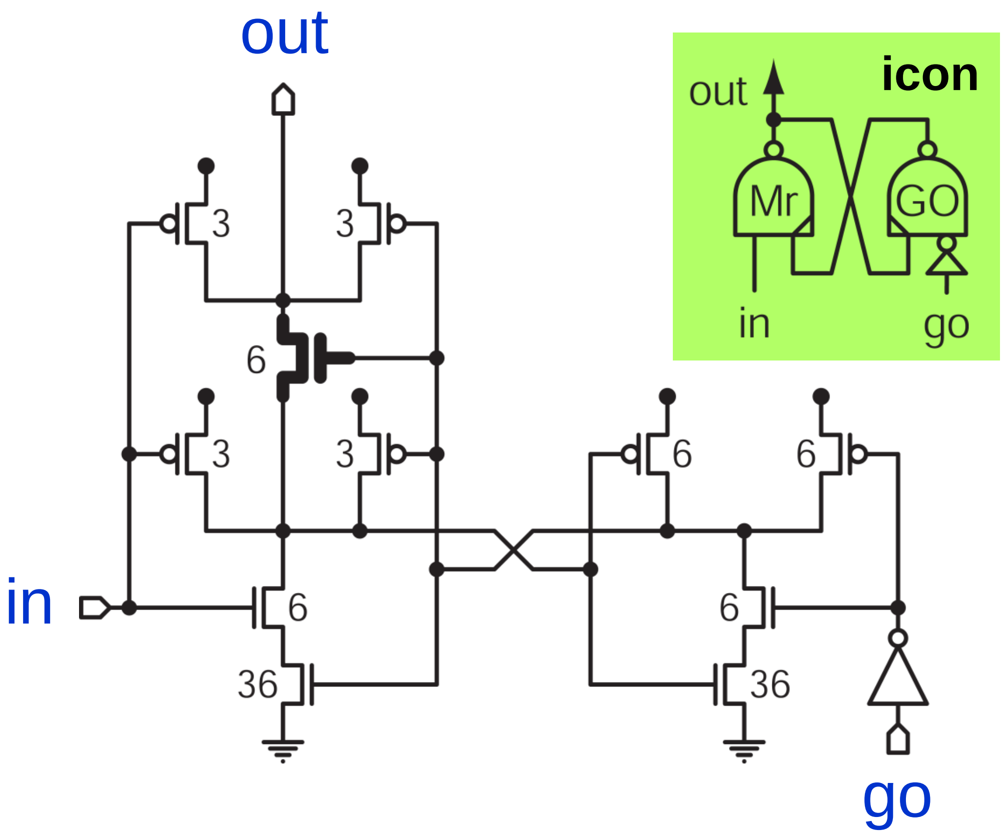
Testing a burst of data at speed



MrGO

MrGO: dedicated action control

- go is high (GO) – start in to out
- go is low (🚫) – stop or freeze in to out
- arbiter for safe stop – "proper stopper"
- scan chain delivers go signals



out
3
3
6
3
3
6
6
in
6
6
36
36
go
icon
out
Mr
GO
in
go