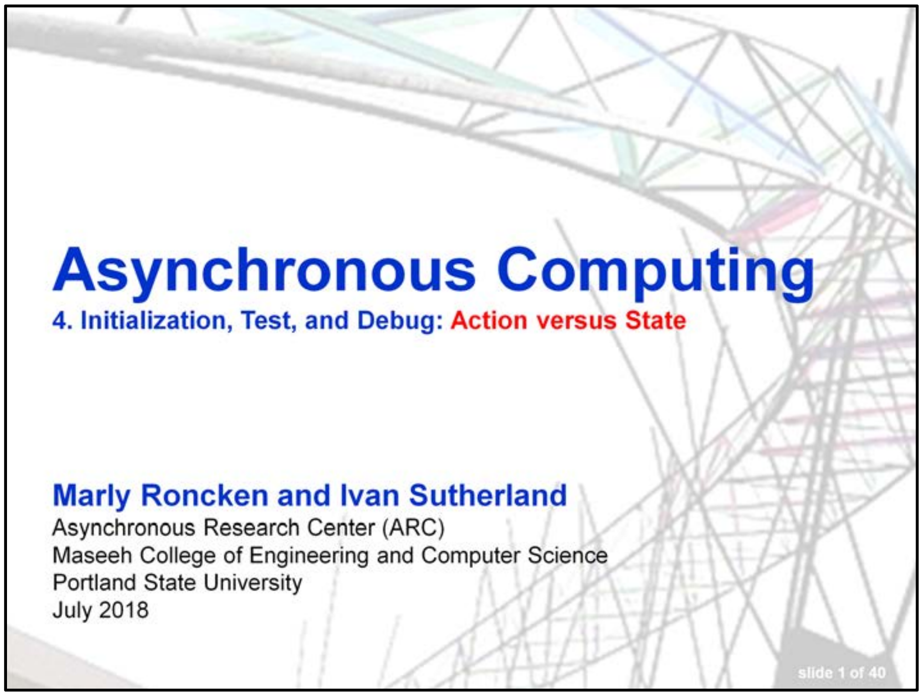


Asynchronous Computing

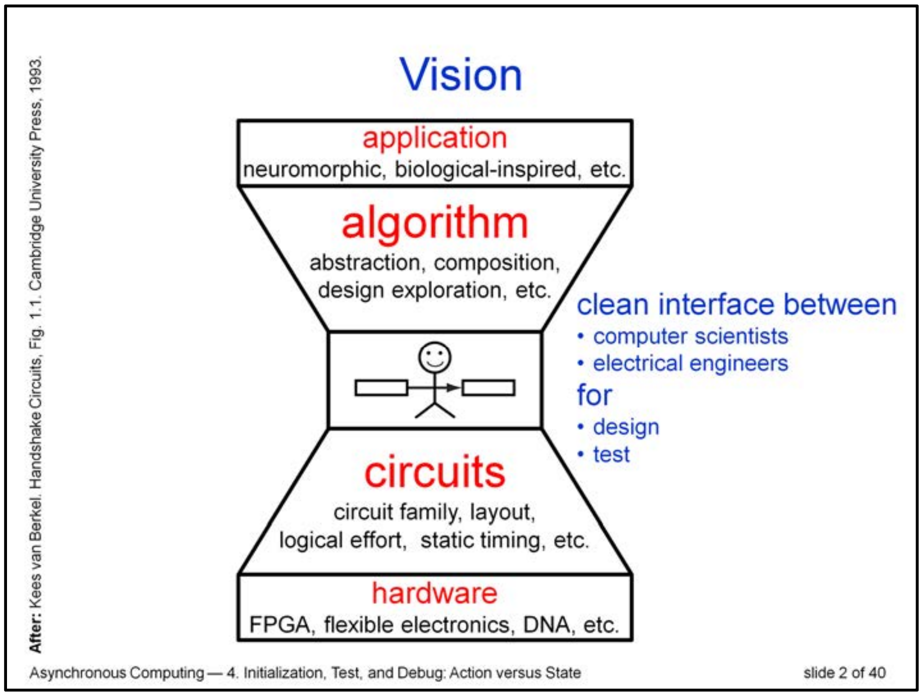
4. Initialization, Test, and Debug: Action versus State

Marly Roncken and Ivan Sutherland
Asynchronous Research Center (ARC)
Maseeh College of Engineering and Computer Science
Portland State University
July 2018

slide 1 of 40

This is part 4 of our ShanghaiTech Lecture on Asynchronous Computing.

We will show how we separate - from the ground up - action from state,
and how both are equally important
to initialize, test, and debug asynchronous systems.



Vision

application
neuromorphic, biological-inspired, etc.
algorithm
abstraction, composition, design exploration, etc.
circuits
circuit family, layout, logical effort, static timing, etc.
hardware
FPGA, flexible electronics, DNA, etc.

clean interface between
- computer scientists
- electrical engineers

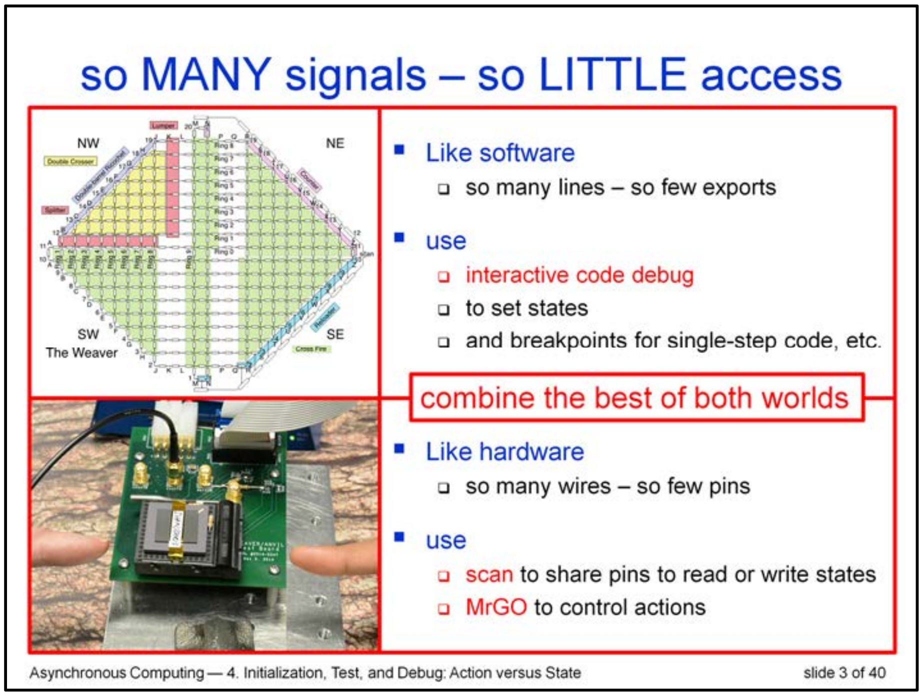
for
- design
- test

After: Kees van Berkel. Handshake Circuits, Fig. 1.1. Cambridge University Press, 1993.

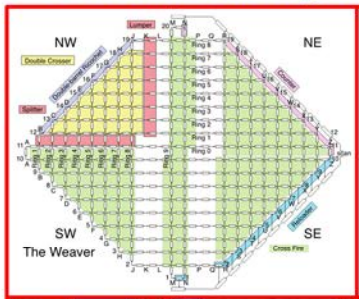
This is a reminder of our bigger vision.

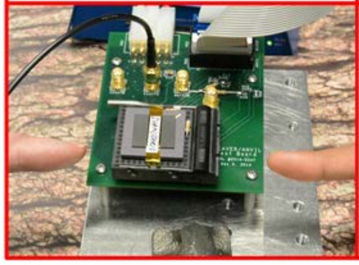
The link-joint model provides a clean interface
between computer scientists and electrical engineers
so they can communicate
without overloading each other with unnecessary details.

The goal of this talk is to extend that interface
<CLICK>
so it covers not only design
but also initialization, test, and debug.



so MANY signals – so LITTLE access





- Like software
  - so many lines – so few exports
- use
  - interactive code debug
  - to set states
  - and breakpoints for single-step code, etc.

combine the best of both worlds

- Like hardware
  - so many wires – so few pins
- use
  - scan to share pins to read or write states
  - MrGO to control actions

Asynchronous Computing — 4. Initialization, Test, and Debug: Action versus State slide 3 of 40

The top picture shows a link-joint view of our Weaver chip.
The bottom picture shows the packaged chip on its test board.
The key problem with test and debug of software and hardware
is that there are so many signals to test but test access is limited.

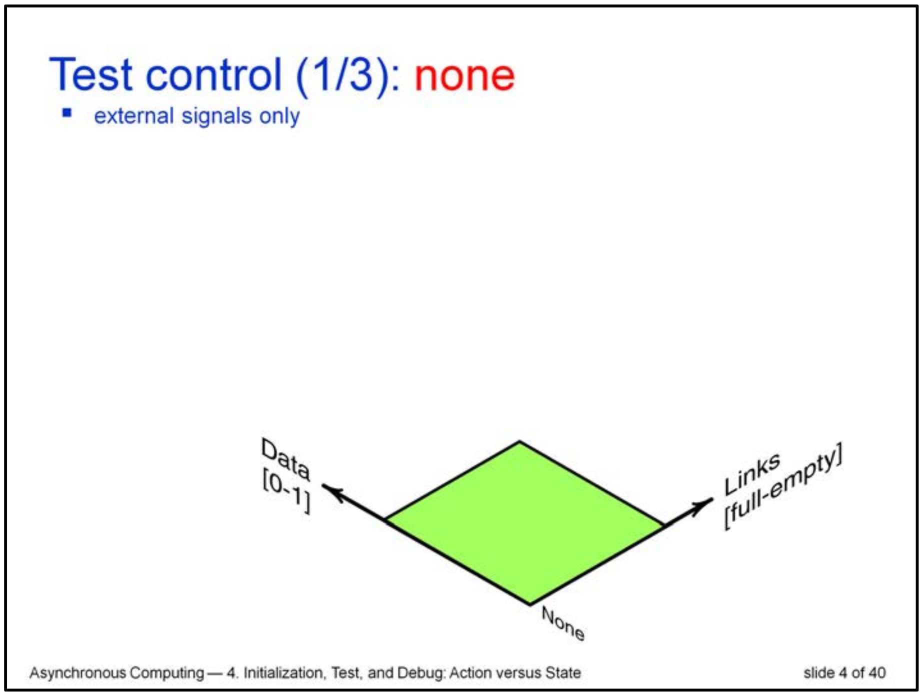
Software code has many lines but few exports.
To debug their code,
programmers use **interactive code debuggers**
to set states
and to set breakpoints to single-step through the code,
or to execute big chunks of code until the next breakpoint,
where they check what happened to the relevant states.

A hardware chip, like the Weaver, has many wires but few external pins.
To debug their chip,
IC designers use a **scan test interface**
to set states through a limited number of pins
so they can read and write states
while they single-step through the chip operations.

We have added a separate control mechanism, called **MrGO**,
to start and stop individual actions.

<CLICK>

The combination of scan and MrGO
forms the basis of a very powerful code and silicon debugger,
which combines the best of both worlds.
It can be used in design as well as in test.



Test control (1/3): none

- external signals only

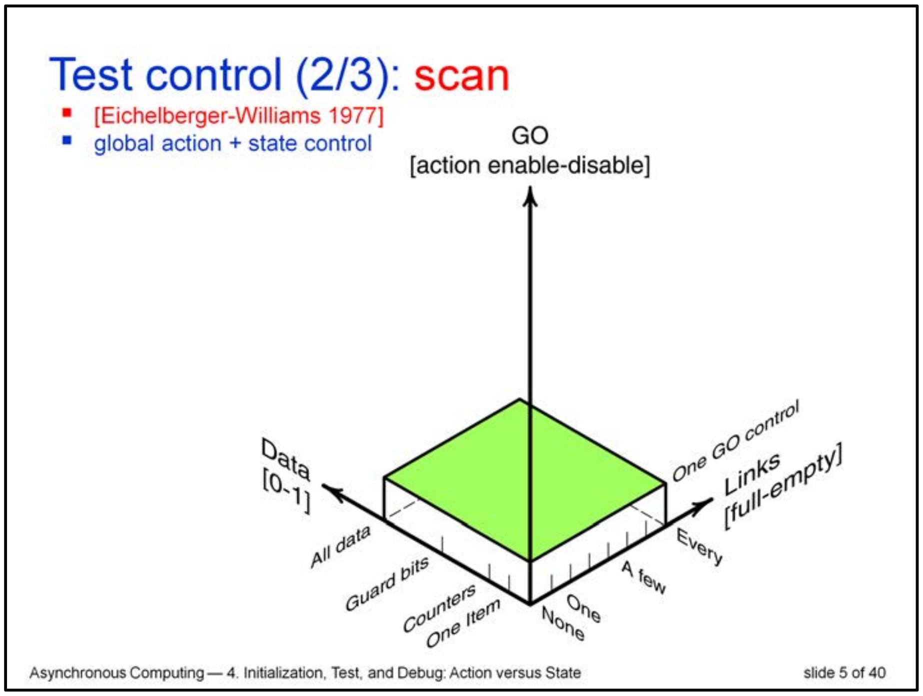
Data [0-1]
Links [full-empty]
None

At the bottom of this slide you see
what a circuit looks like from the point of view of testing,
when there's no test control built into the circuit.

A tester knows there's data and there are links,
and that a data bit has a binary value of type 0 or 1,
and that a link has a binary value of type full or empty.
BUT...
the tester can see only external data and links;
it cannot see internal data and links.

For most circuits, most of the green area you see is internal to the chip.
The tester can control and observe the green area only INDIRECTLY...
by running the circuit as is.

That is a big problem, and one that's not new to modern chip designs.
It was already a big problem in 1977, when at IBM,
Edward Eichelberger and Thomas Williams presented a test control
solution that became a standard in the test world.
[next slide]



Test control (2/3): scan

- [Eichelberger-Williams 1977]
- global action + state control

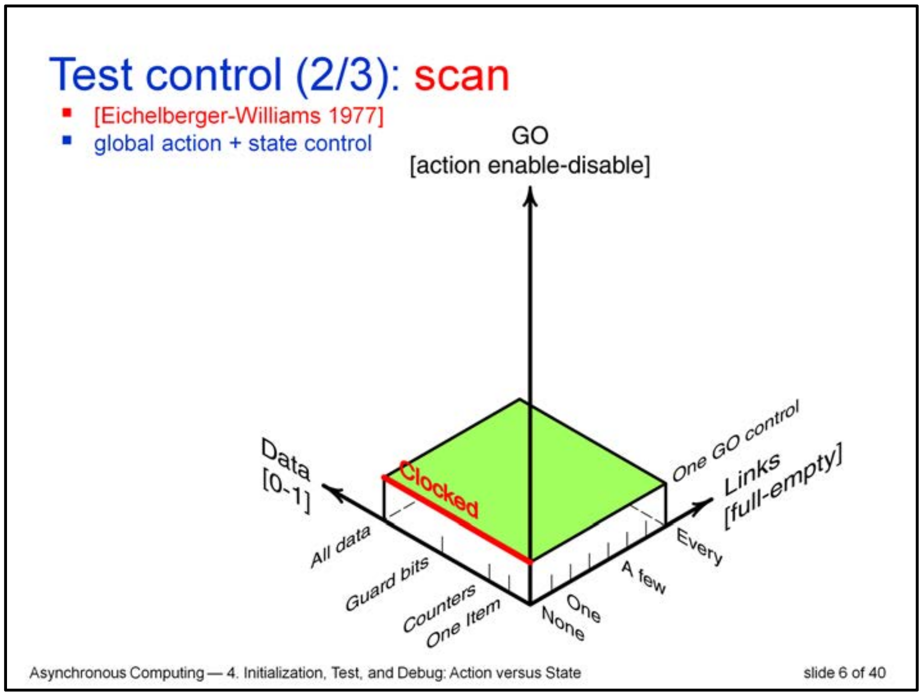
Today it's know as scantest or simply scan.
Scan refers to special test circuitry to control the state of a circuit, globally.
It comes with a special mode, to turn the circuit action on and off,
while the tester reads or writes data bits and links in the circuit.

In the test world, that special mode is often called a test mode.
In this presentation, I call it a GO signal.

The traditional scan test approach comes with one GO signal.
  * When it's high, or enabled, it allows the circuit to act as usual.
  * When it's low, or disabled, it stops all circuit action.

On this page, you see the new 3D vision of the tester.
This is what the tester sees of a circuit with scan.
The difference in what the tester sees with scan and without it is enormous.
Look at what the tester could see before
  [BACK one SLIDE]
and look at what it sees now
  [FORWARD to this slide]

Scan is an eye opener.
Where the tester was blind before, now it can see any internal signal it wants.
  * It can read, and write any single data bit, counters, guards - you name it.
    The data bits are shown here on the left axis in this 3D test space.
  * It can read and write any link.
    The links are lined up here on the right axis.
  * And it can enable or disable the global circuit action,
    through the one and only GO control bit here on the vertical axis.



Test control (2/3): scan

- [Eichelberger-Williams 1977]
- global action + state control

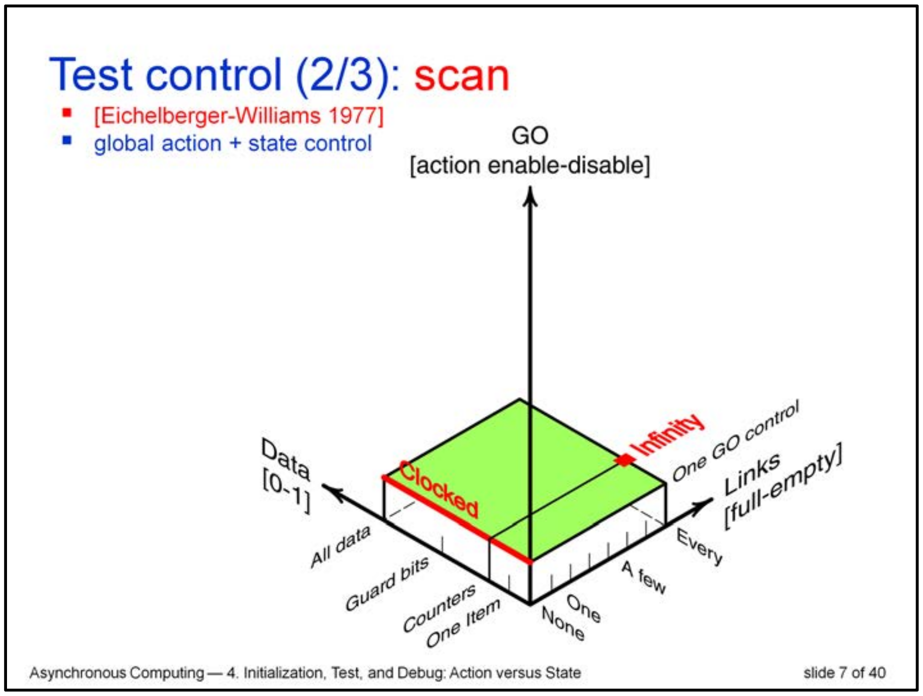
The red line indicates what a tester can control
in a globally clocked synchronous circuit - with scan.

It can read and write anything on this red line.
But it might not want to do that.

To save area or power,
it might want to control only the most important data.
For instance, the counters and the guards,
and perhaps one data item to load and unload data
for indirect control to the remaining data.

If only part of the data are scanned, it's called a partial scan test.
If all data are scanned, it's called a full scan test.

Note that the red line ignores the links
- no links are scanned for reading or writing.
This is because clocked systems don't use links.
Links with their full-empty communication interfaces
are unique to self-timed design.



Test control (2/3): scan

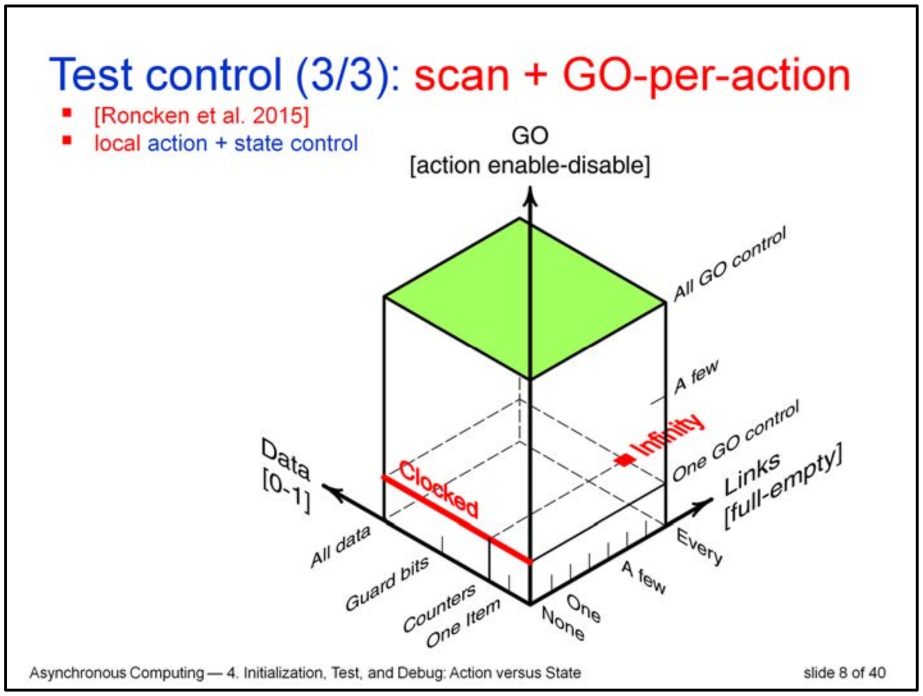
- [Eichelberger-Williams 1977]
- global action + state control

Asynchronous Computing — 4. Initialization, Test, and Debug: Action versus State slide 7 of 40

In 2008, the VLSI group at Sun Microsystems Laboratories,
led by Ivan, made Infinity.

Infinity is a self-timed chip design that uses scan.
It lives here in this 3D test space.
* It scans every link,
* all the counters (for throughput analysis),
* and it can load and unload one data item
  - to set data items beyond its control
  - using the normal self-timed operation of the circuit.

This scan design was sufficient to test and debug Infinity.



Test control (3/3): scan + GO-per-action

- [Roncken et al. 2015]
- local action + state control

To see what we're still missing requires that we recognize
that **a self-timed system isn't about global action.**
The actions of a self-timed system are self-generated
and widely distributed in both space and time.
It's both these properties , self-generation AND distribution,
that traditional scan test and clocked design fail to support.

* Self-generation cannot be mimicked by the rigid ticks of a global clock.
The ticks of a self-timed circuit vary and adapt.
Only by embracing their variety and adaptivity
can we support at-speed test and debug
of self-timed circuits and systems.

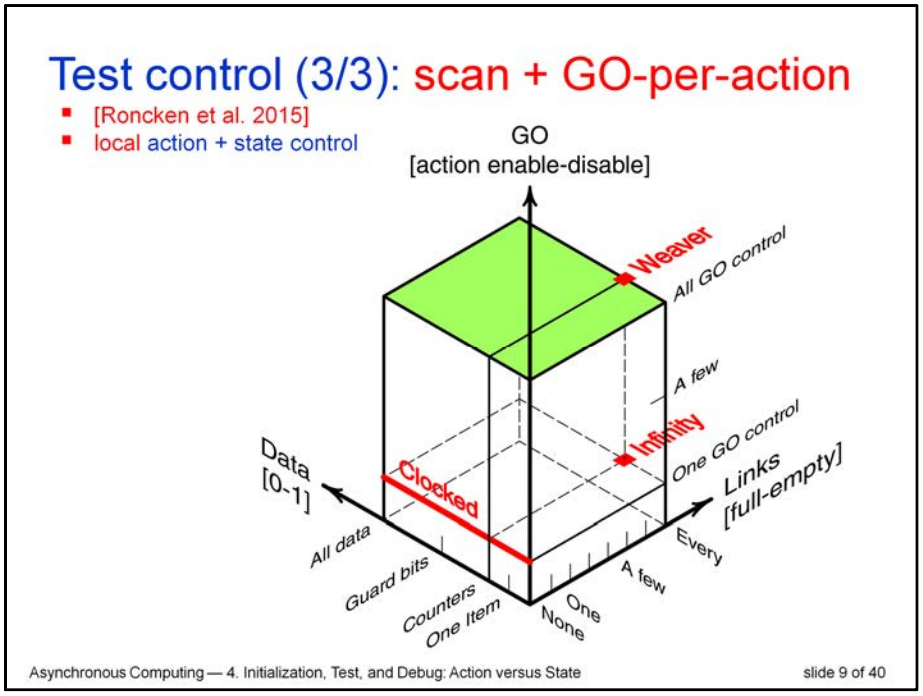
This implies that we control each and every action, individually.
Instead of global control, we make the control local.

For distributed actions, local control makes good sense too.

So, instead of one GO control for all,
we take two, or three, or a few ...or all.

"All GO control" here at the top of the vertical axis
indicates that there is an individual GO control signal for
each and every local action.

The idea is that by locally enabling or disabling actions, we can carve out
test paths, test tunnels, and various test areas for self-timed actions.
This supports initialization, parallel testing,
and single-step, multi-step test, and at-speed test and debug.

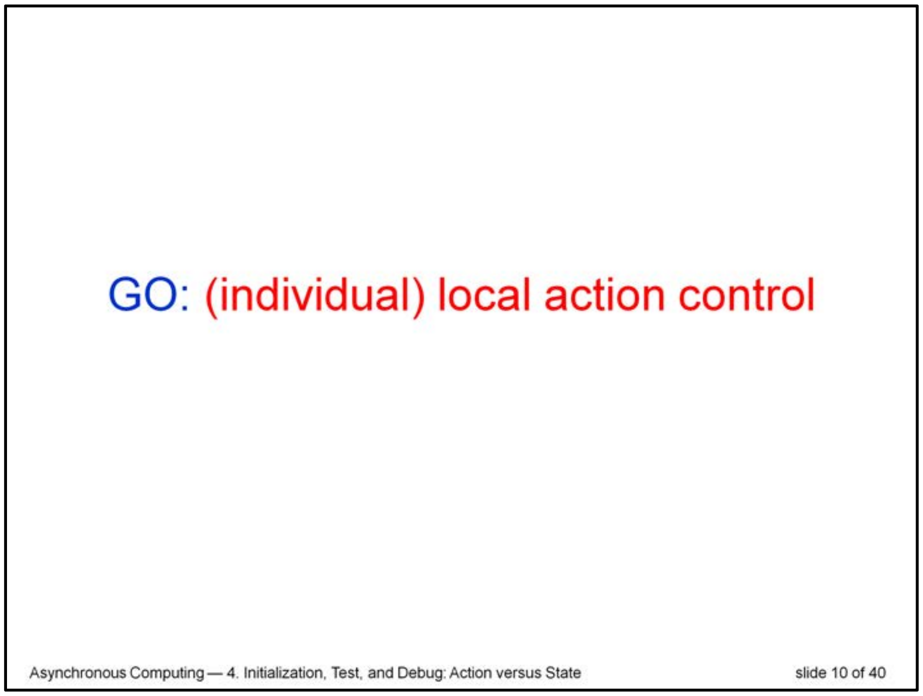


Test control (3/3): scan + GO-per-action

- [Roncken et al. 2015]
- local action + state control

Our latest chip, the Weaver, lives <HERE>:
it scans all GO control signals,
it scans every full-empty state,
it scans the counters,
and it can load and unload one Data item.

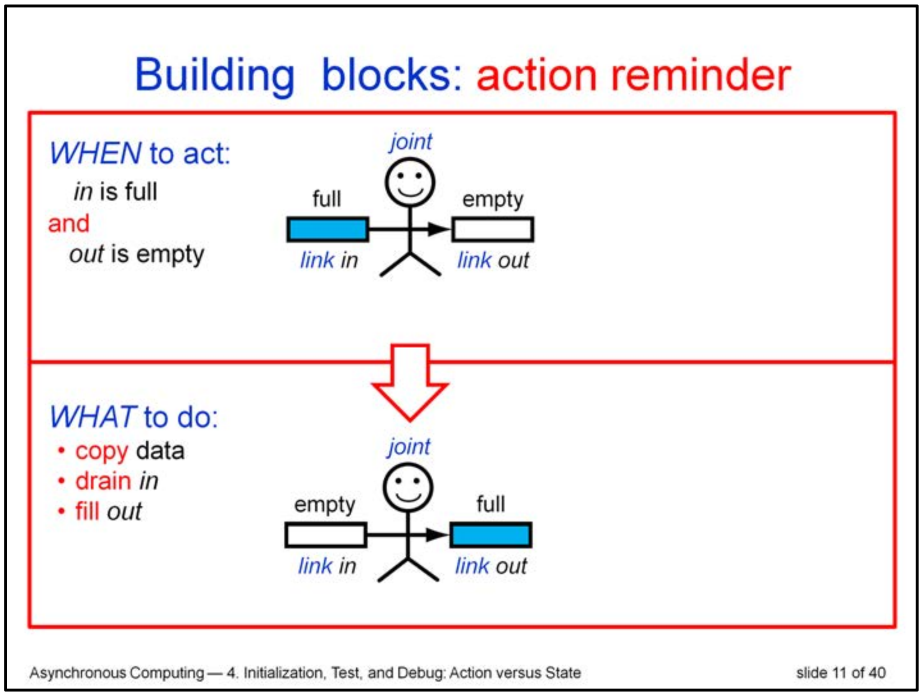
The test examples later in this talk come from the Weaver.



GO: (individual) local action control

Asynchronous Computing — 4. Initialization, Test, and Debug: Action versus State slide 10 of 40

So, how do we GO there?



Building blocks: action reminder

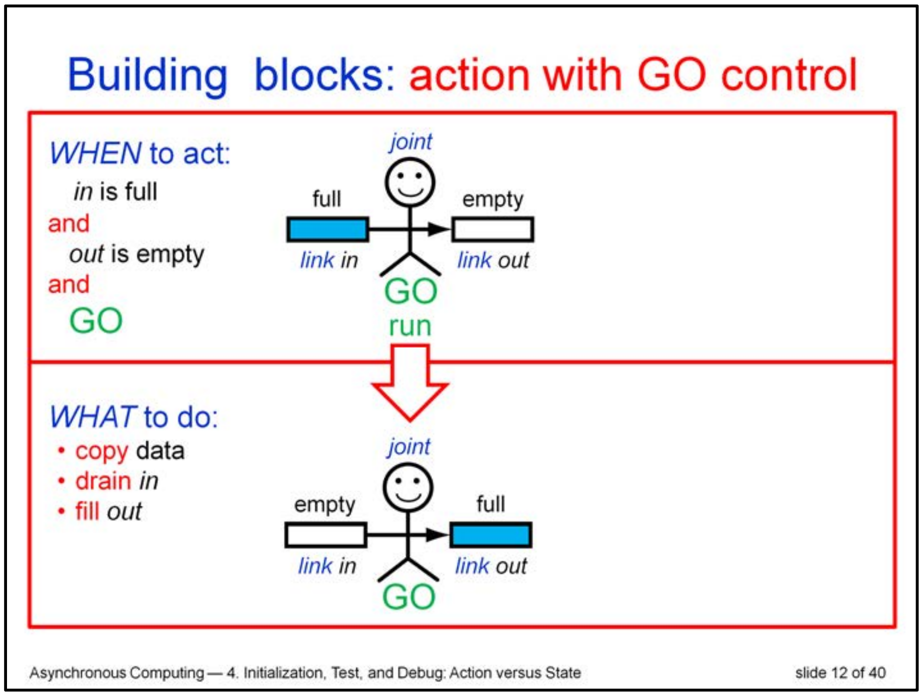
WHEN to act:
in is full
and
out is empty

WHAT to do:
- copy data
- drain *in*
- fill *out*

The first step is to recognize self-timed actions.
Here is a reminder of what a self-timed action looks like.

When link *in* is full (blue) **AND** link *out* is empty (white)
copy the data
drain *in*
and fill *out*.

NOTE:
We target computation and flow control actions.
These are in the joints.
The actions in a link are partly shared with the neighboring joint
- e.g. store data and full is shared with the filling joint,
- e.g. store empty is shared with the draining link,
and partly a (link) transport delay away
from becoming shared with the joint on the other end,
when the other end selects a guard
with the resulting transported change in full-empty,
and acts upon it.
From a semantics point of view: joints act, links transport.
We need to fine-tune our terminology to better reflect this.



Building blocks: action with GO control

WHEN to act:
in is full
and
out is empty
and
GO

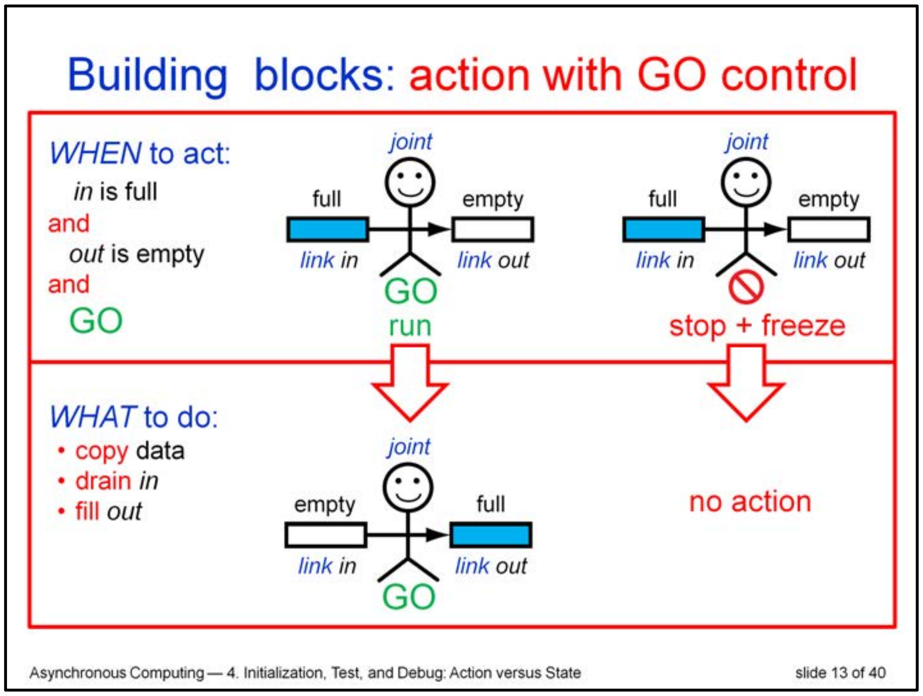
WHAT to do:
- copy data
- drain *in*
- fill *out*

joint
full
empty
link in
link out
GO
run
joint
empty
full
link in
link out
GO

To control this action
we add a GO control signal to the **and-function** in the **when** part of the action.
We leave the **what** part as is.

Now:
When link *in* is full (blue) **AND** link *out* is empty (white) **AND** GO is enabled
copy the data
drain *in*
and fill *out*.

When GO is enabled, the action runs as before.



Building blocks: action with GO control

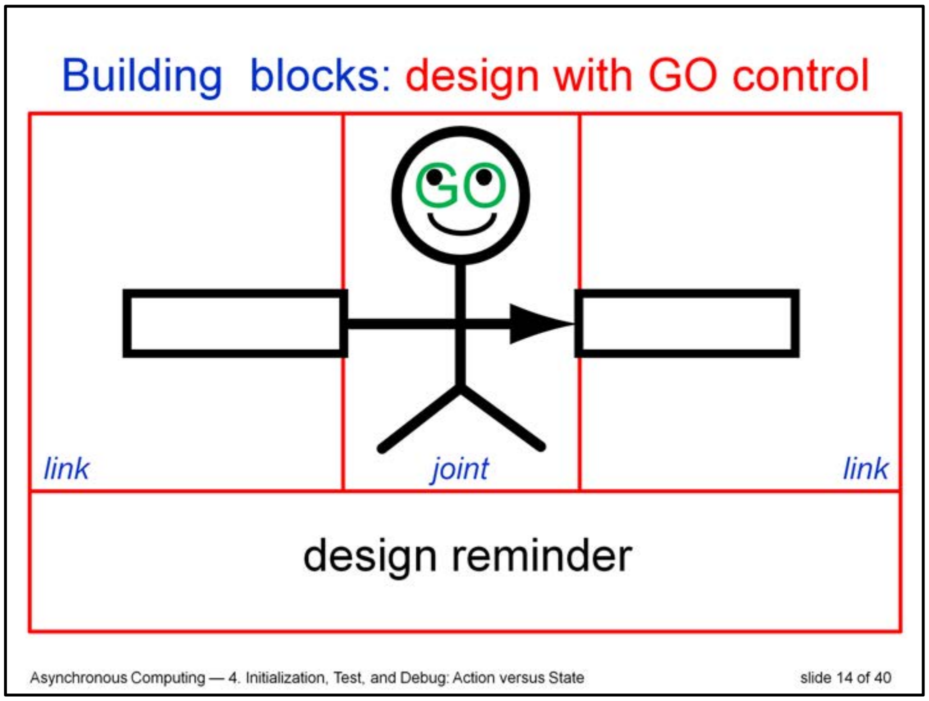
WHEN to act:
- *in* is full

and
- *out* is empty

and
- GO

WHAT to do:
- copy data
- drain *in*
- fill *out*

But when GO is **dis-**abled
the action stops and freezes.

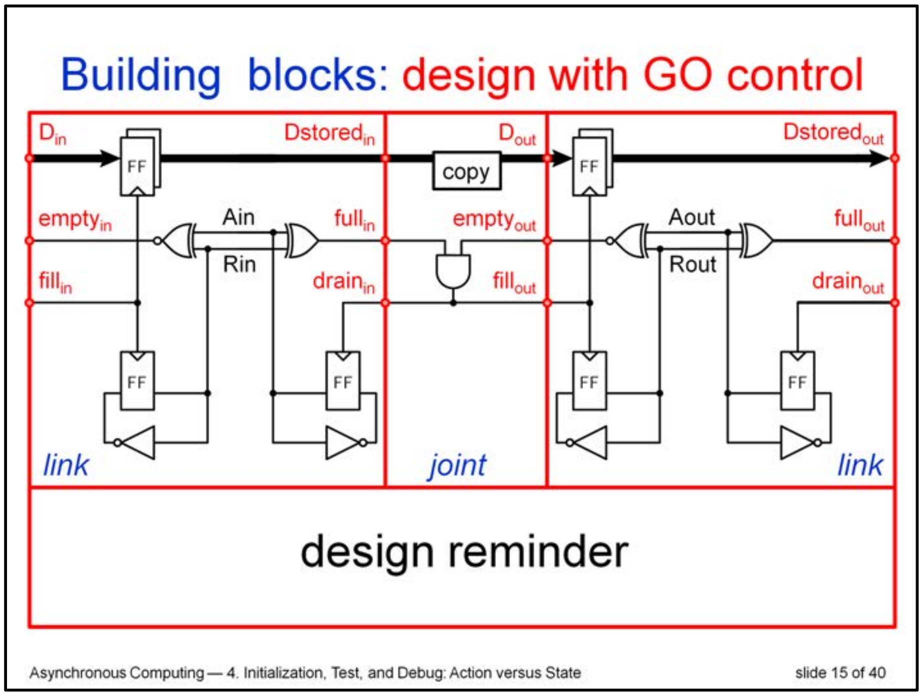


Building blocks: design with GO control

Where do we add this GO control?

Well ... the and-function is located in the joint
so the joint gets the GO control.

Here is a reminder of what a joint looks like
<CLICK to NEXT SLIDE>



Building blocks: design with GO control

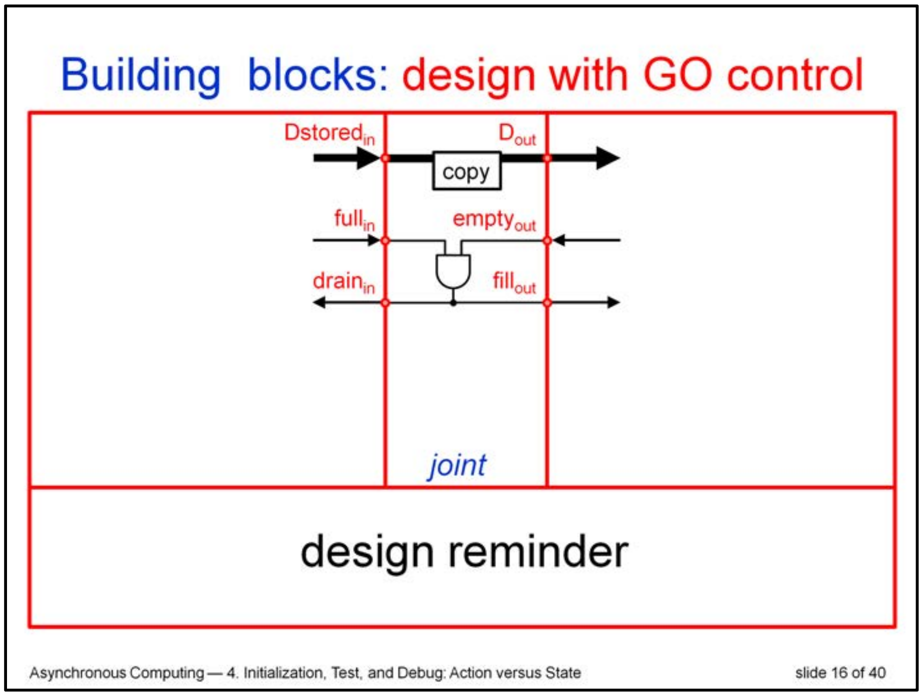
Asynchronous Computing — 4. Initialization, Test, and Debug: Action versus State slide 15 of 40

This is one of the FIFO circuits from my earlier talk
about links and joints.

It has a joint and two Click links.

Because the GO control is in the joint, and not in the links,
we can ignore the links here.

Let's ignore the links.
[next slide]



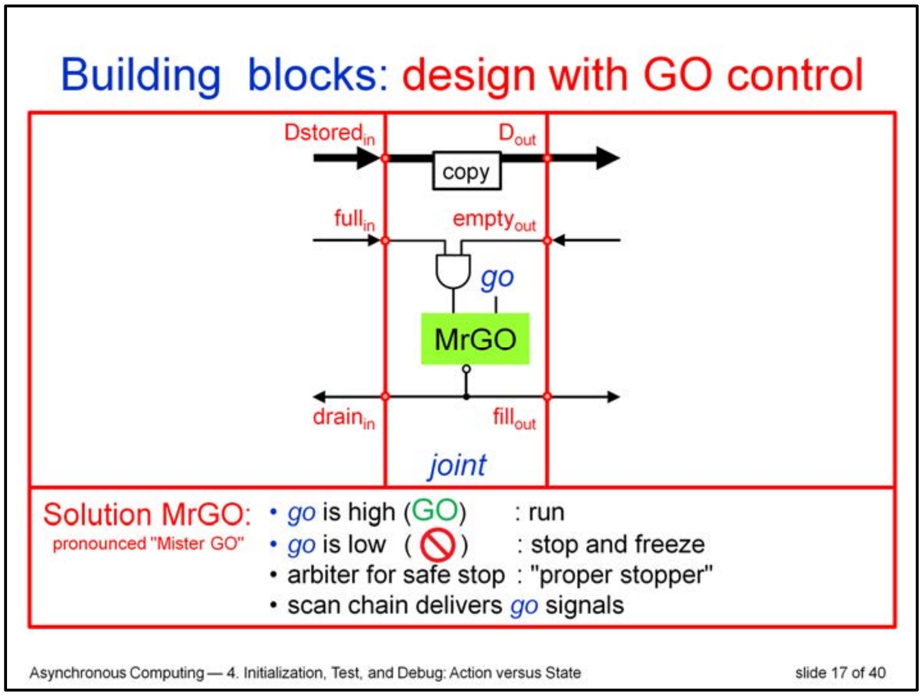
Building blocks: design with GO control

design reminder

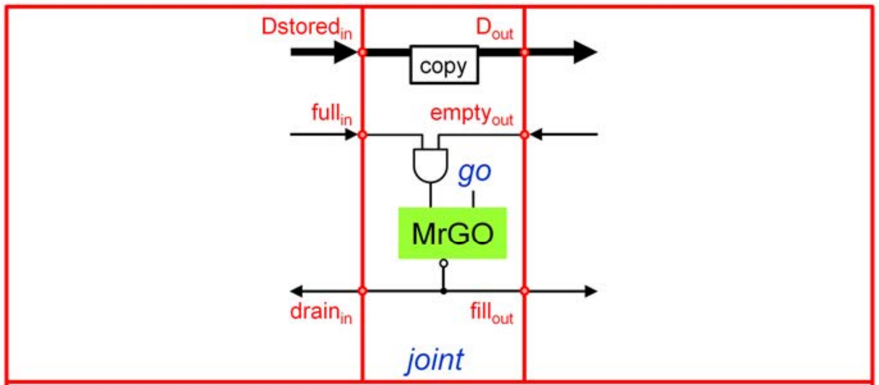
Asynchronous Computing — 4. Initialization, Test, and Debug: Action versus State   slide 16 of 40

So what do we have now?
We have an AND gate
plus some combinational logic in the datapath
to copy the data from in to out.

And that's it... for the FIFO.



Building blocks: 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

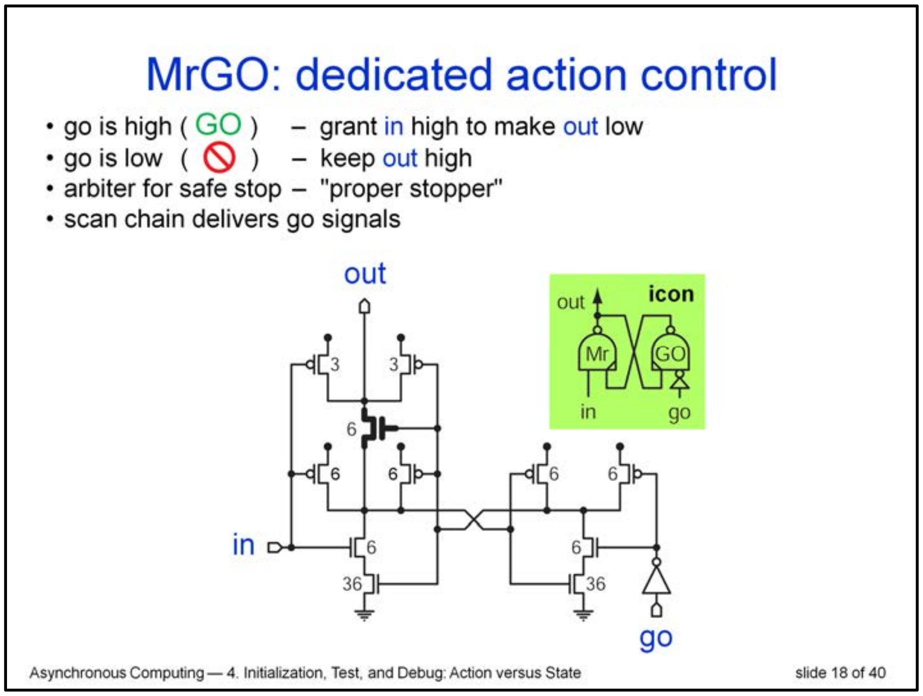
We are now ready to add GO control.
We add GO control by adding a GO control signal after the AND gate,
and we call that signal *go.*

The *go* signal comes with its own arbiter - the green box.
so we can safely stop self-timed actions in full flight.
The green box is called **"Mister GO" (MrGO).**
Do you remember Ivan's talk about arbitration (part 3)?
MrGO arbitrates between a low GO signal, to stop the action,
and a high input signal coming from the AND gate, to continue the action.
The arbiter either stops the action, if the low go signal wins,
or it continues the action if the high AND signal wins.

If the action continues because the high AND signal won the race,
then the output of MrGO will go low
and its inversion used here will go high
and the joint will drain in and fill out... **and then it will self-reset.**
As soon as it self-resets, the AND gate will go low,
and the go signal will grab the arbiter and stop the next action.
MrGO is a non-blocking arbiter that's fair to the loser.

To summarize: The green box is called **"Mister GO" (MrGO).**
<CLICK>
When *go* is high
MrGO unfreezes the joint action, and allows it to run as usual.
When *go* is low
MrGO acts like a **proper stopper** and will stop and freeze the joint action safely.

We use the scan chain to deliver the individual *go* signals.



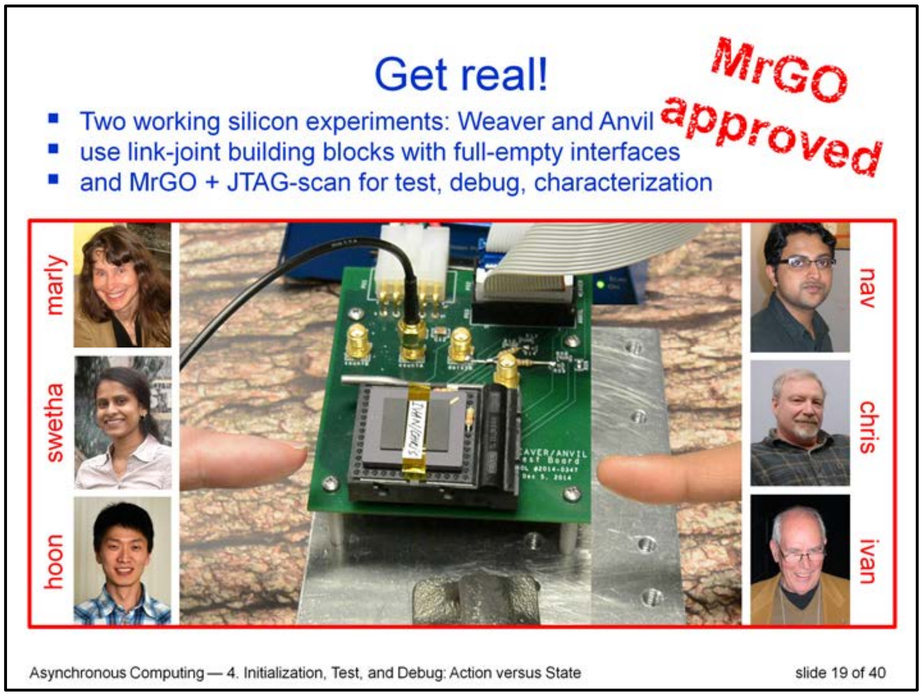
MrGO: dedicated action control

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

This is a transistor level implementation of MrGO.

Do you recognize the arbiter circuit from Ivan's talk about arbiters?

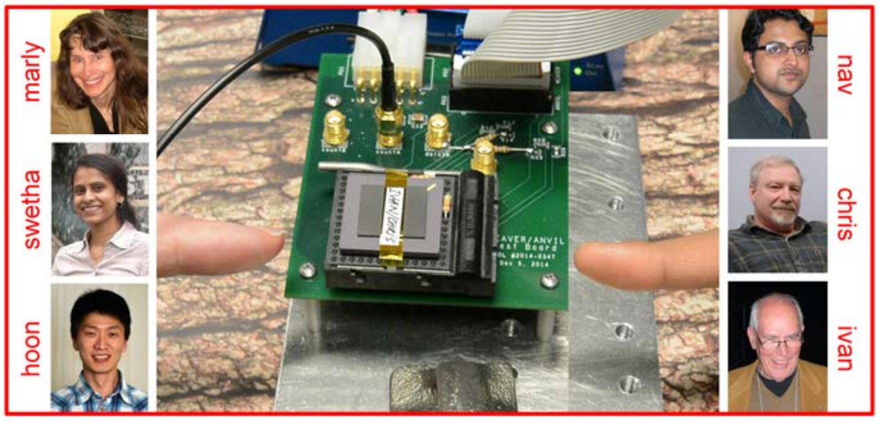
MrGO arbitrates between
a high *in* signal to make *out* low
and a low GO signal to keep *out* high.



Get real!

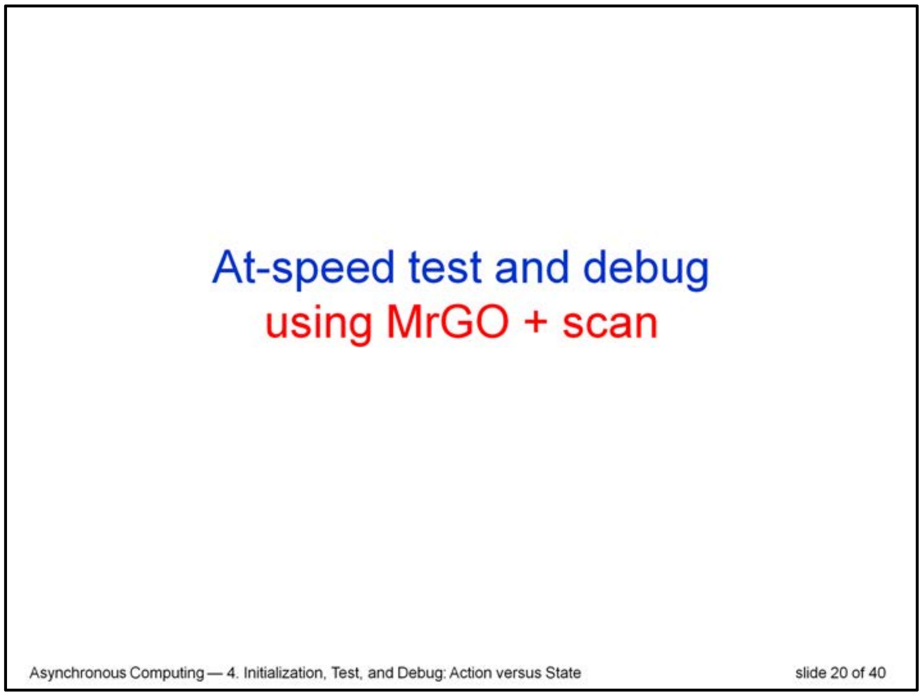
- Two working silicon experiments: Weaver and Anvil
- use link-joint building blocks with full-empty interfaces
- and MrGO + JTAG-scan for test, debug, characterization

MrGO approved
marly
swetha
hoon
nav
chris
ivan



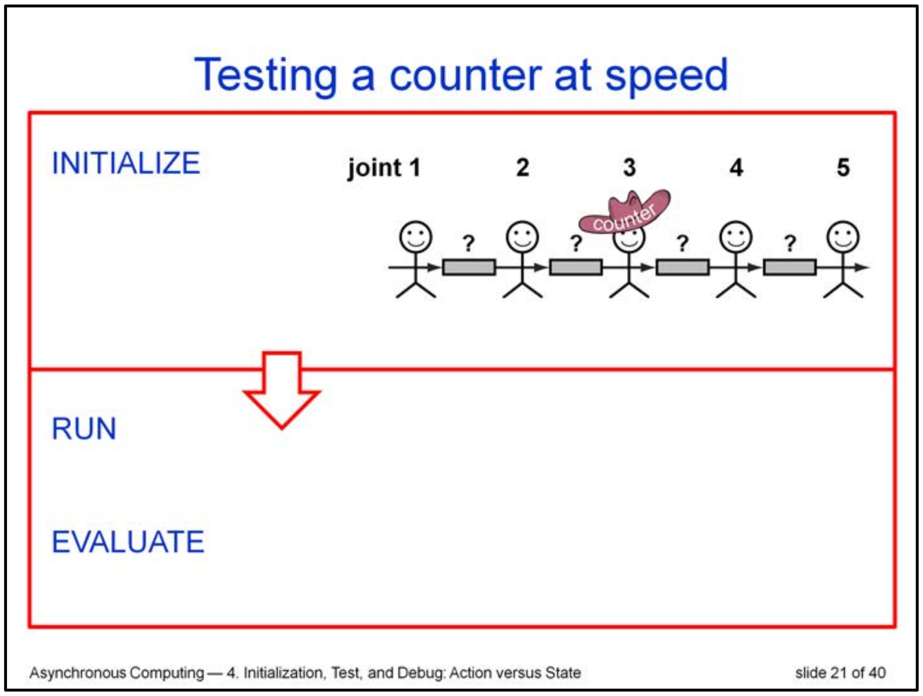
We have two working silicon experiments, called Weaver and Anvil.
Both use links and joints with full-empty interfaces
and MrGO with an industry-standard JTAG scan interface for test, debug, and characterization.

<CLICK for "MrGO approved" STAMP>
Both the Weaver and Anvil have passed all tests
that we, our students, and our visitors have thrown at them.



At-speed test and debug using MrGO + scan

The proof of the pudding is in the eating.
So, let's do a test.



Testing a counter at speed

INITIALIZE

RUN

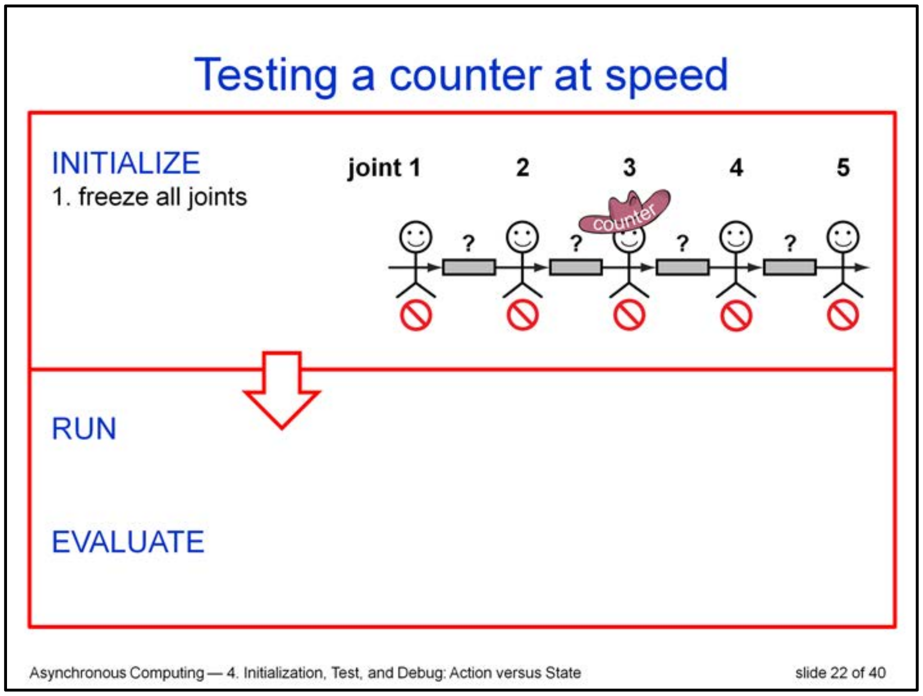
EVALUATE

Here is a FIFO with 5 joints.
Joint number 3 has a cowboy hat - that's a counter.

We will test the counter at speed.

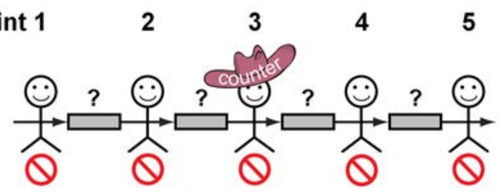
We can do this in three steps, called
Initialize, run, and evaluate.

NOTE:
for proper alignment, the text has been hidden by making it white,
and uncovered in the next few slides by making it black.



Testing a counter at speed

INITIALIZE
1. freeze all joints



RUN

EVALUATE

Asynchronous Computing — 4. Initialization, Test, and Debug: Action versus State

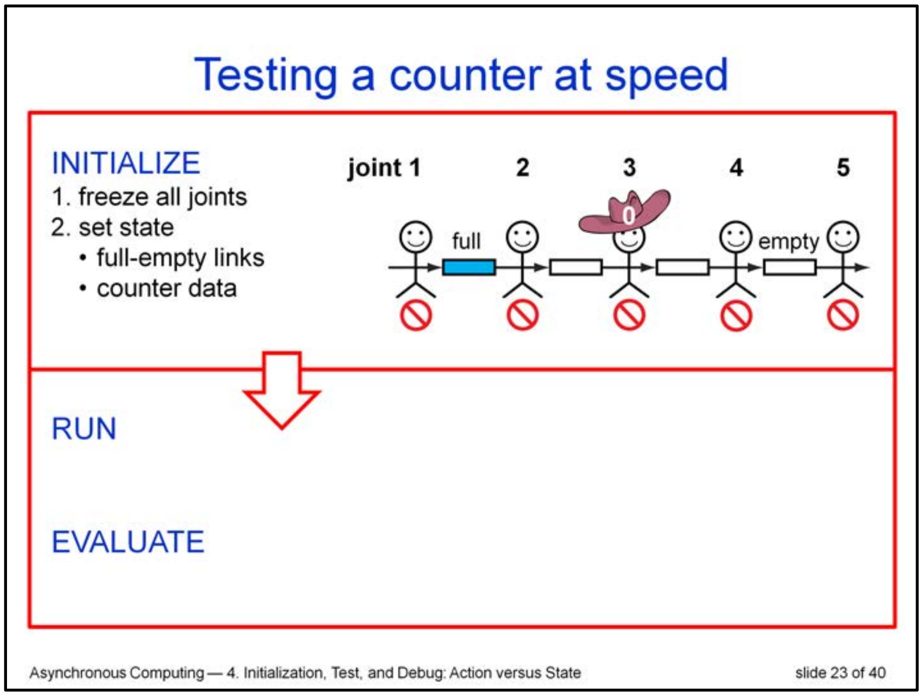
slide 22 of 40

To initialize the system, we first freeze all the joints.

We do this by making all the *go* signals low
as indicated by the red stop signs.

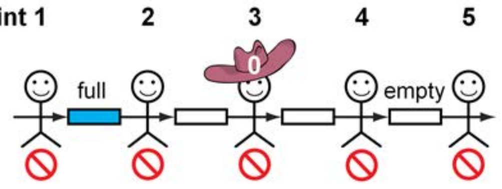
We use a scan chain to initialize the go signals.

This will stop **every** action in the FIFO.



Testing a counter at speed

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



RUN

EVALUATE

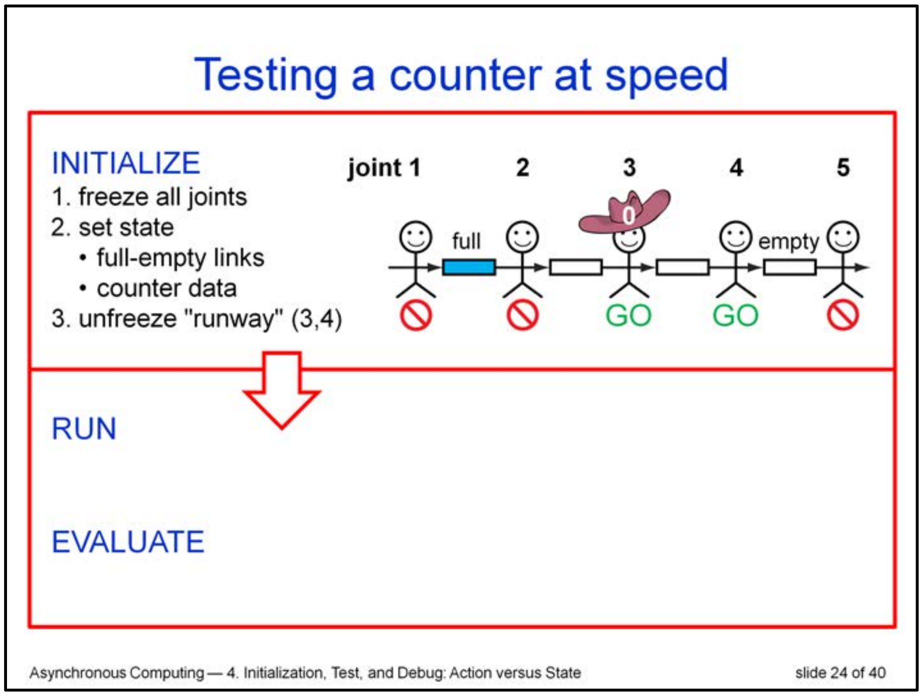
Asynchronous Computing — 4. Initialization, Test, and Debug: Action versus State   slide 23 of 40

Next, we set the state.

To run one DATA item through this FIFO,
we make the first link full
and the other links empty.

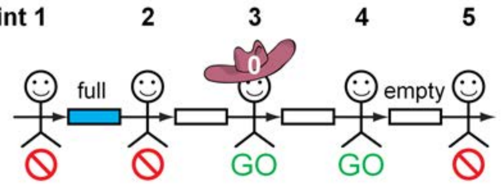
We also set the counter.
In this case, we choose to make the counter zero.

Again, we use a scan chain to initialize the links and the counter.



Testing a counter at speed

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



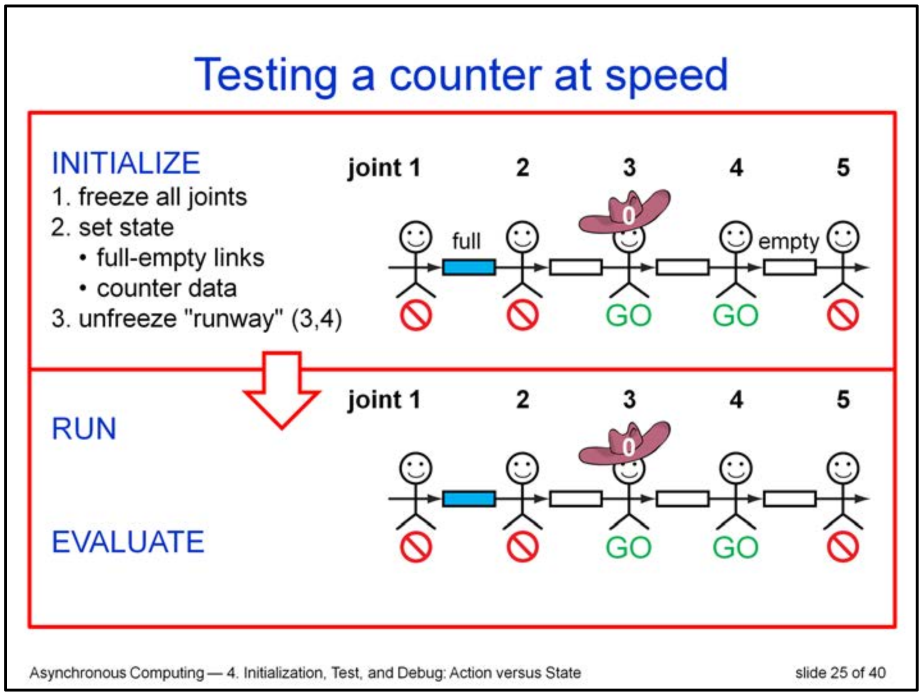
RUN

EVALUATE

Asynchronous Computing — 4. Initialization, Test, and Debug: Action versus State slide 24 of 40

Then we open the landing runway by unfreezing joints 3 and 4.

We use a scan chain to set these two GO signals.



Testing a counter at speed

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

RUN

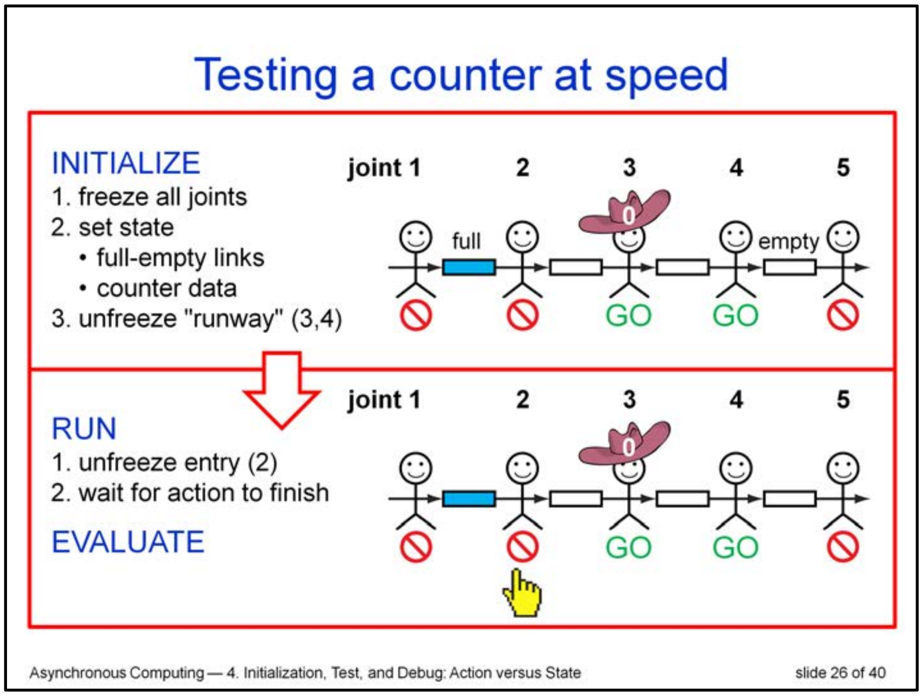
EVALUATE

joint 1
2
3
4
5
full
empty
0
GO
GO
joint 1
2
3
4
5
0
GO
GO

Asynchronous Computing — 4. Initialization, Test, and Debug: Action versus State slide 25 of 40

And that's our initial state.

Note that we kept the first and last joints frozen.
This confines the test setup.
Other test inputs cannot get in,
and our test results cannot escape.



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

The test is now ready for take-off.
We permit it to take off by making the *go* signal of joint number 2 high.
That's the *go* signal at the hand-cursor.

We call joint 2 **"the gate keeper"**.

We use the scan chain to enable this GO signal.

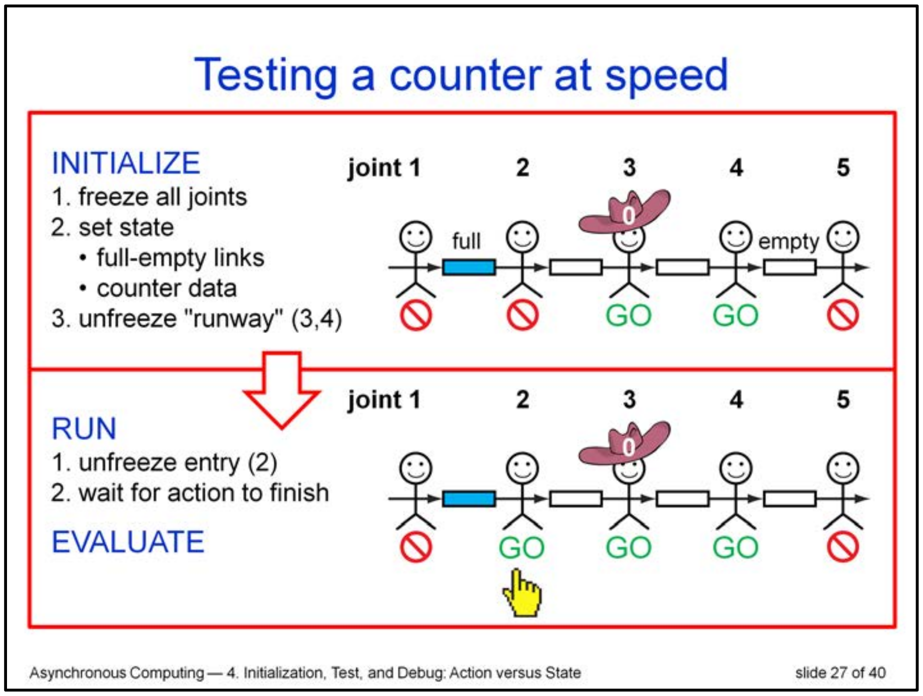
As soon as the go signal at the hand-cursor is high,
the data in the blue (full) link will make three moves, going left to right,
and increase the counter value by 1.

This will all happen at speed,
without any interaction from me.

At speed in this presentation is 1 second per move.
It will take 3 seconds for the blue data to move from left to right.

In the Weaver, which is 40nm CMOS, each move is ONLY 100 picoseconds.
So the blue data will zip through in 300 picoseconds.

Here we GO!



Testing a counter at speed

INITIALIZE

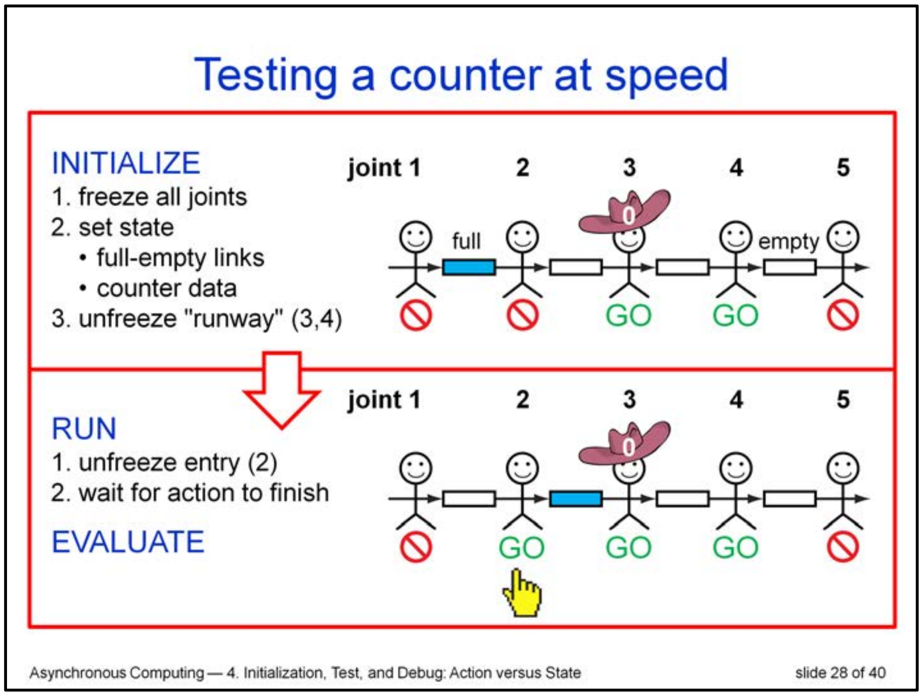
1. freeze all joints
2. set state
   - full-empty links
   - counter data
3. unfreeze "runway" (3,4)

RUN

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

EVALUATE

<SELF-TIMED>



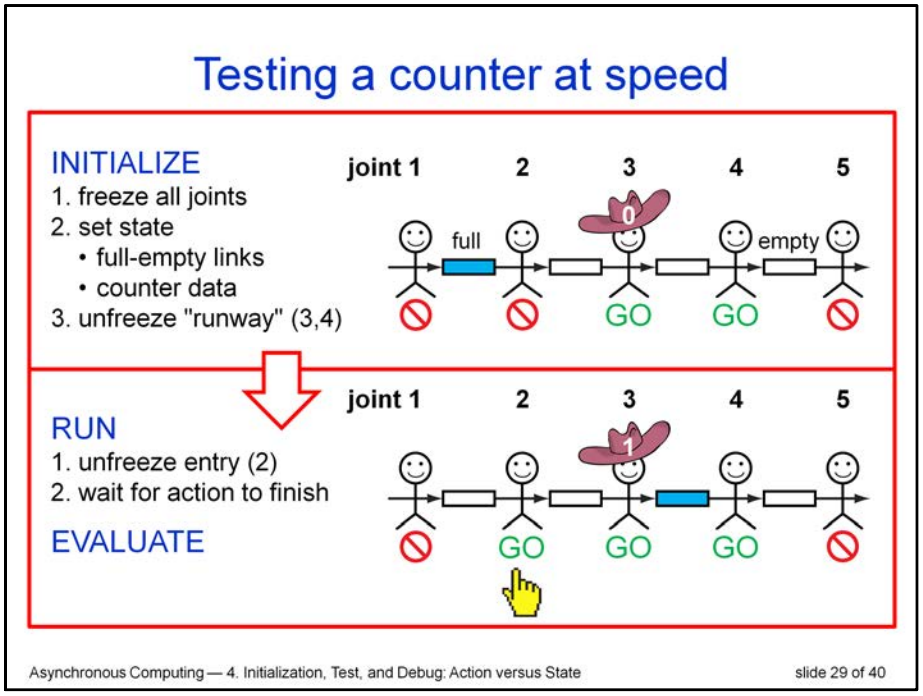
Testing a counter at speed

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

<SELF-TIMED>



Testing a counter at speed

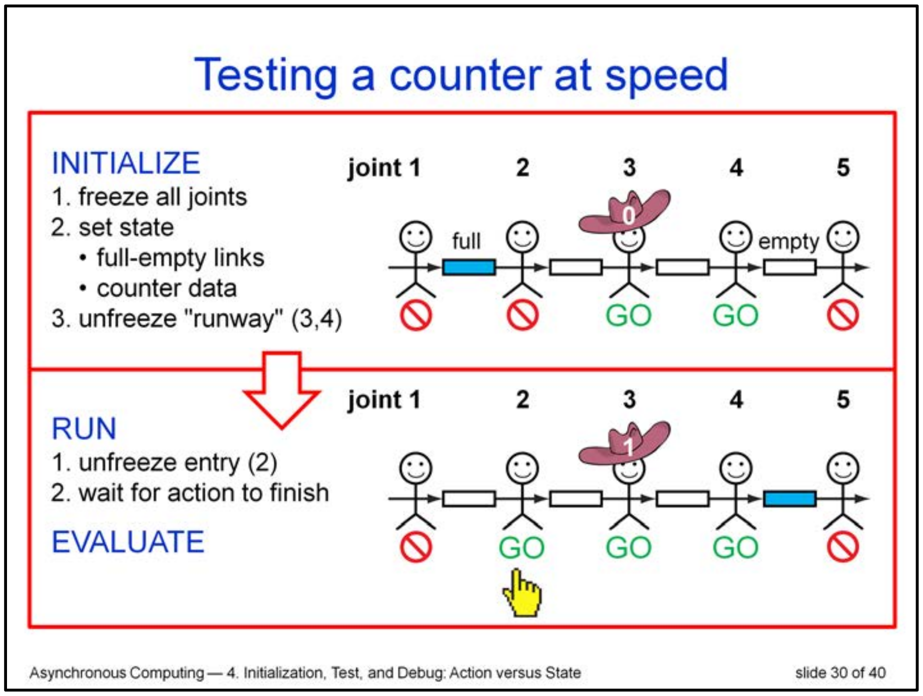
INITIALIZE
1. freeze all joints
2. set state
   - full-empty links
   - counter data
3. unfreeze "runway" (3,4)

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

EVALUATE

joint 1
2
3
4
5
full
empty
0
GO
GO
joint 1
2
3
4
5
1
GO
GO
GO

<SELF-TIMED>



Testing a counter at speed

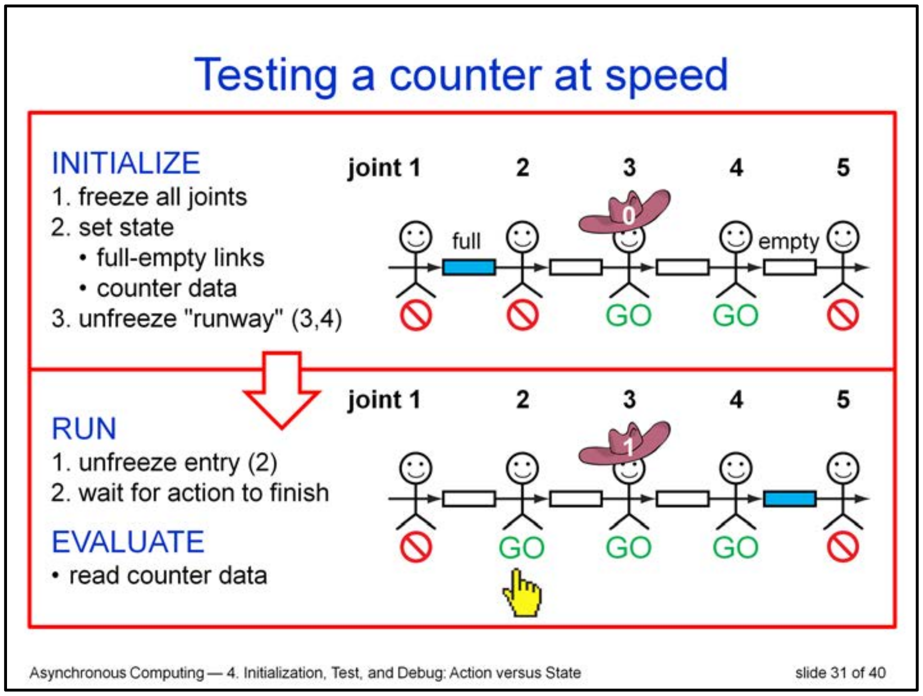
INITIALIZE
1. freeze all joints
2. set state
   - full-empty links
   - counter data
3. unfreeze "runway" (3,4)

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

EVALUATE

joint 1
2
3
4
5
full
0
empty
GO
GO
joint 1
2
3
4
5
1
GO
GO
GO

<SELF-TIMED>



Testing a counter at speed

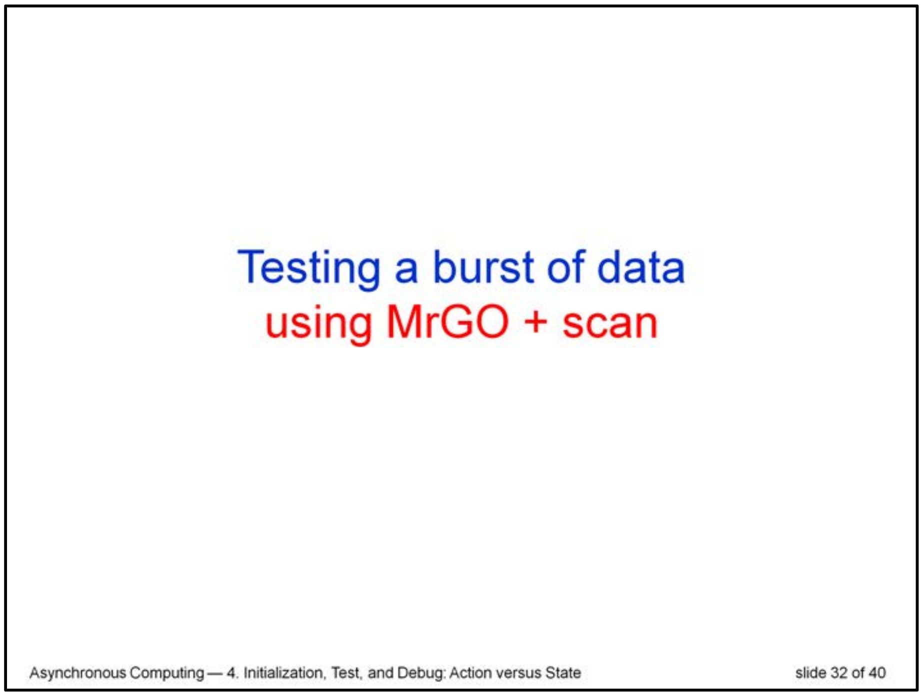
INITIALIZE
1. freeze all joints
2. set state
   - full-empty links
   - counter data
3. unfreeze "runway" (3,4)

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

EVALUATE
- read counter data

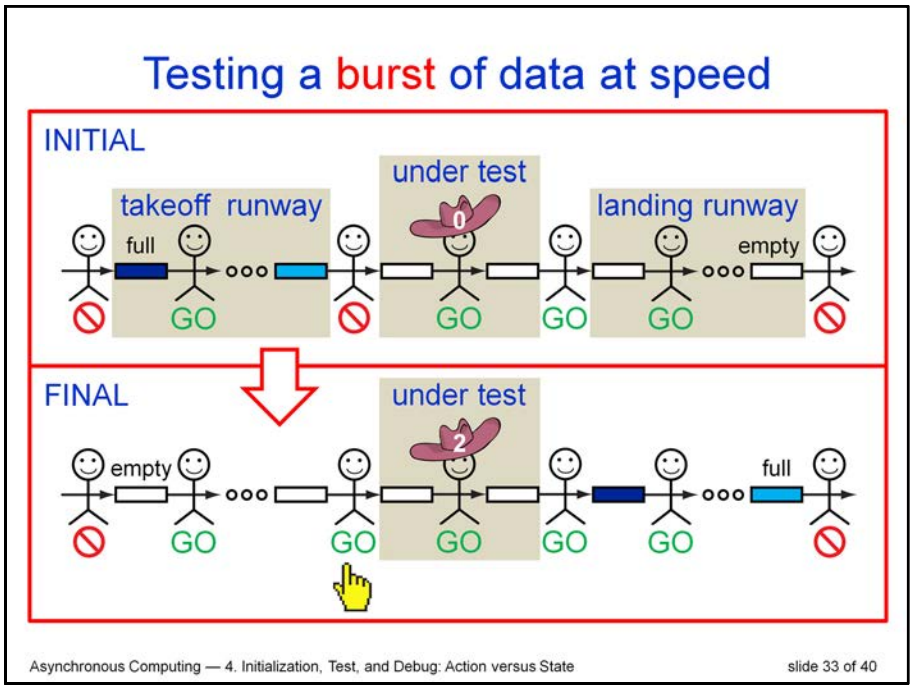
After 300 picoseconds, we scan the counter data out to validate that it's now 1.

AND SO IT IS !!!



Testing a burst of data
using MrGO + scan

We can easily extend the previous test
to test a burst of data items
at speed
through the counter.



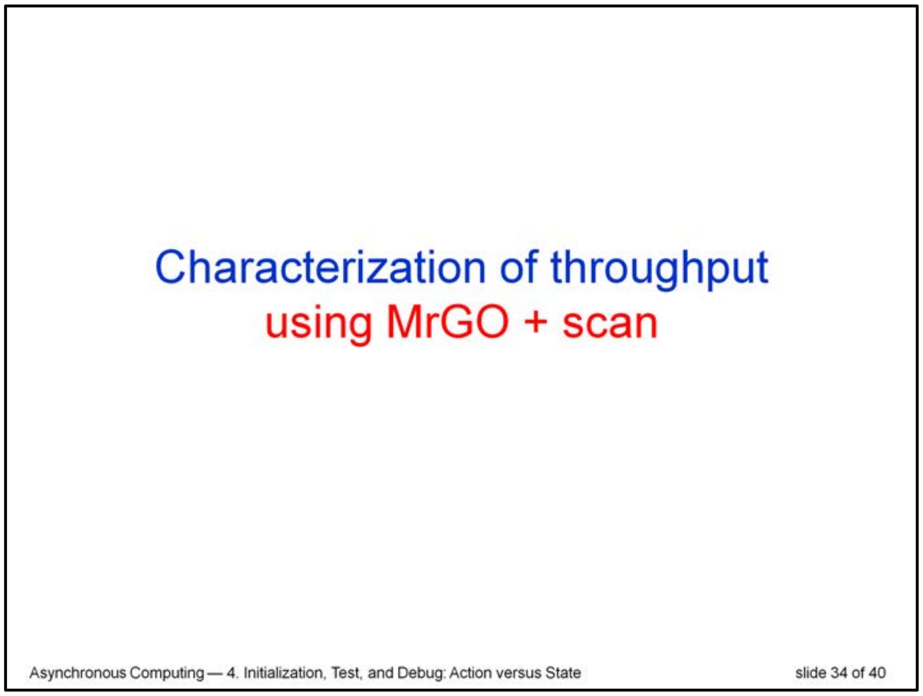
Testing a burst of data at speed

At the top-left, we set up a take-off runway
with as many full links with data items as we want in the burst.

At the top-right, we set up a landing runway
with as many empty links as are needed
to store the results generated in this test.

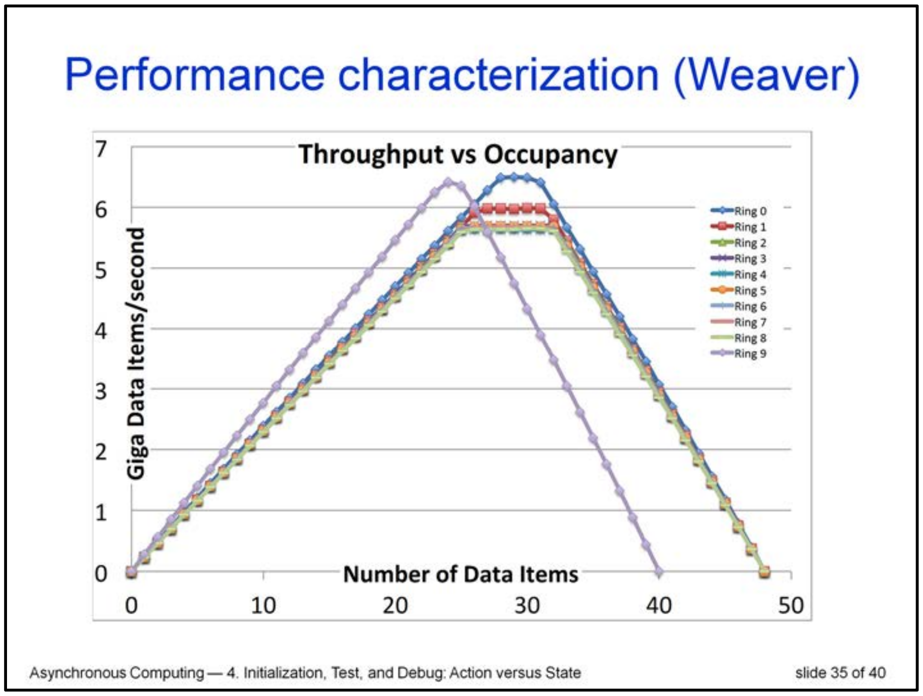
As before, we kept the first and last joints in each runway frozen.
to confine the test setup,
so other test inputs cannot get in,
and our test results cannot escape.

Then we unfreeze the gate keeper,
which is the joint with the handcursor.
We let the circuit run its course,
and then scan out the data captured by the landing runway.



Characterization of throughput using MrGO + scan

Remember the canopy graphs that we use to characterize throughput?



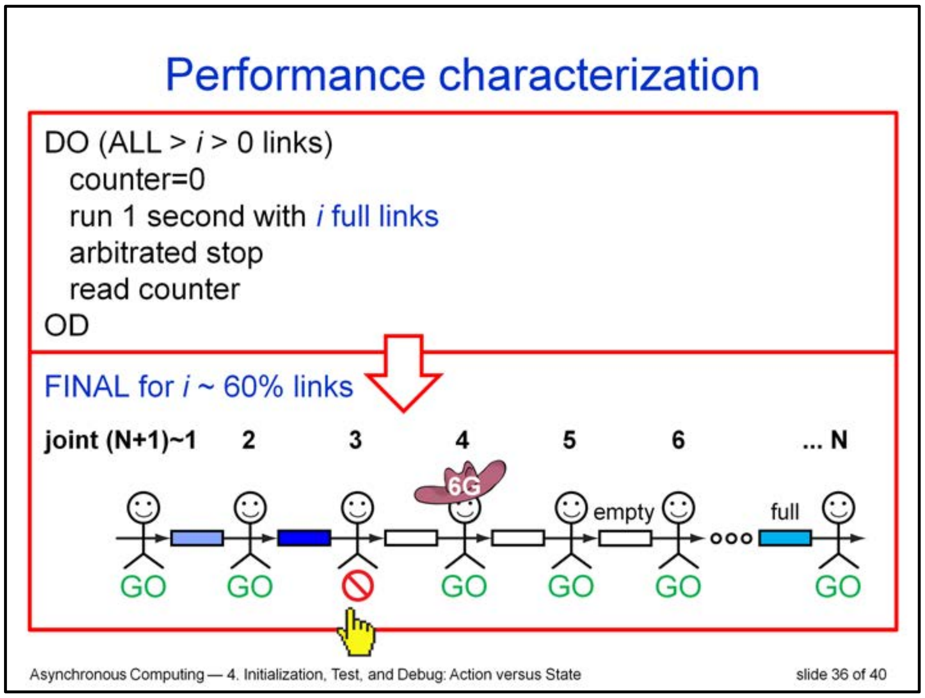
Performance characterization (Weaver)

Here is a reminder.

These are the canopy graphs **measured from the Weaver chip.**
We've seen them earlier in my talk about link and joint building blocks.
The graphs show the throughput for the various ring-FIFOs in the Weaver.

The horizontal axis shows the number of full links in the ring.

The vertical axis shows the throughput measured as the number of GigaDataItems per second counted by each ring counter.



Performance characterization

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

FINAL for *i* ~ 60% links

Asynchronous Computing — 4. Initialization, Test, and Debug: Action versus State   slide 36 of 40

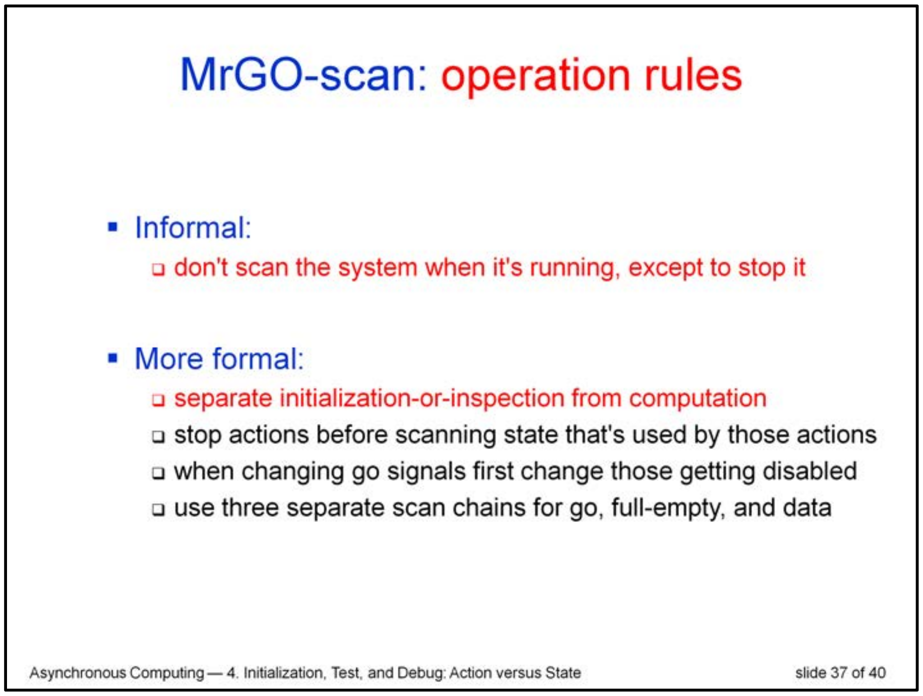
This is how we measure the throughput for those canopy graph:

- We initialize the ring counter to zero.
- We make *i* links full and the other links empty.
- We run the system by enabling the *go* signal of the gate-keeper.
  The gatekeeper is the joint at the yellow handcursor.
- After 1 second we disable that *go* signal.
- Then we read out the counter value.

The link-joint picture on the bottom of the slide
shows that with 60% of the links full
the ring counter counts 6 Giga Data Items in a single second.

Let's go back to the canopy graph
<GO BACK TO PREVIOUS SLIDE>
See: the canopy graphs of the eight rings that go through the crossbar
show a throughput of 6 Giga Data Items per second when the ring is 60% full,
that is when 28-29 of the links are full.
<FORWARD TO THIS SLIDE>

NOTE:
no boundary joints here - only a gatekeeper,
because the experiment is already contained by the ring.



MrGO-scan: operation rules

- Informal:
  - don't scan the system when it's running, except to stop it
- More formal:
  - separate initialization-or-inspection from computation
  - stop actions before scanning state that's used by those actions
  - when changing go signals first change those getting disabled
  - use three separate scan chains for go, full-empty, and data

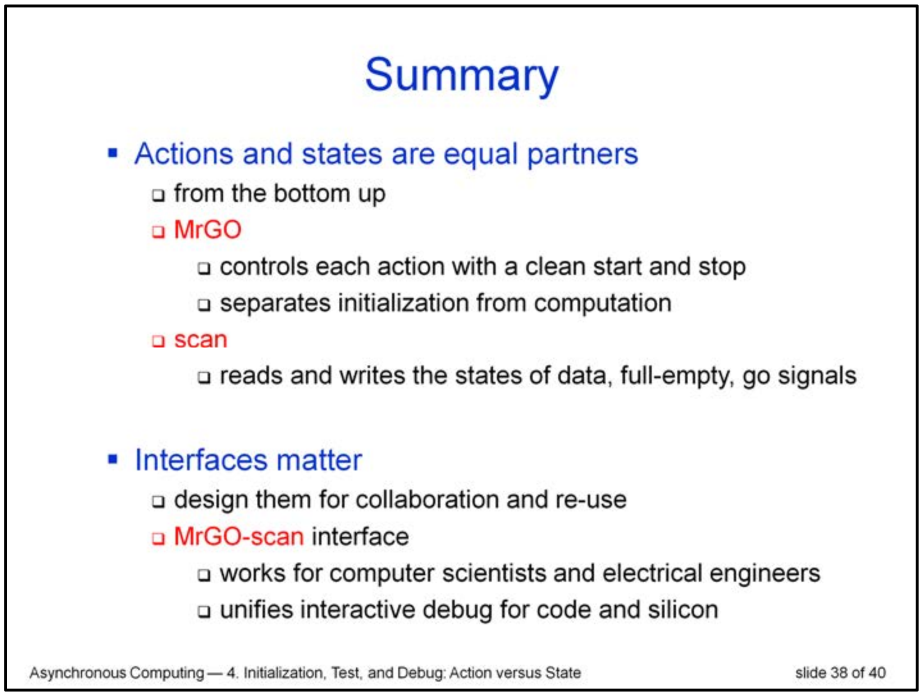
Asynchronous Computing — 4. Initialization, Test, and Debug: Action versus State slide 37 of 40

There are a few rules to make MrGO and scan work together.

The crux is to avoid interference between test and circuit operations.

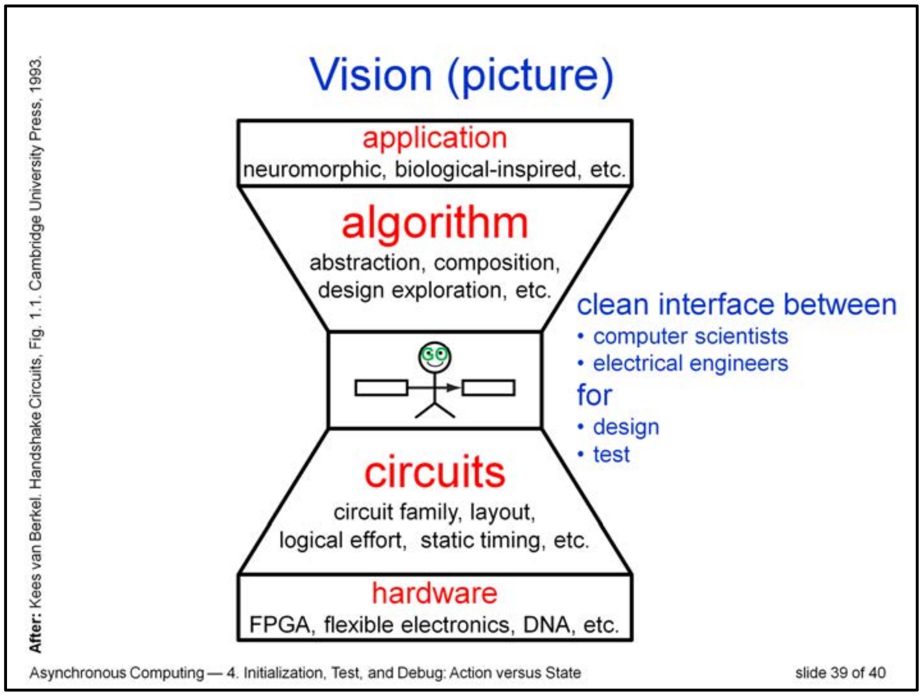
Informally, this boils down to:
- Don't scan the system when it's running, except to stop it.

More formally, it's a good idea to
(1) Stop the actions in a joint before scanning state in or out of links
that are used by those actions.
(2) First change go signals that are being disabled
before you enable go signals.
(3) Use three separate scan chains for go, full-empty, and data.



Summary

- Actions and states are equal partners
  - from the bottom up
  - MrGO
    - controls each action with a clean start and stop
    - separates initialization from computation
  - scan
    - reads and writes the states of data, full-empty, go signals
- Interfaces matter
  - design them for collaboration and re-use
  - MrGO-scan interface
    - works for computer scientists and electrical engineers
    - unifies interactive debug for code and silicon



Vision (picture)

application
neuromorphic, biological-inspired, etc.
algorithm
abstraction, composition, design exploration, etc.
circuits
circuit family, layout, logical effort, static timing, etc.
hardware
FPGA, flexible electronics, DNA, etc.

clean interface between
- computer scientists
- electrical engineers

for
- design
- test

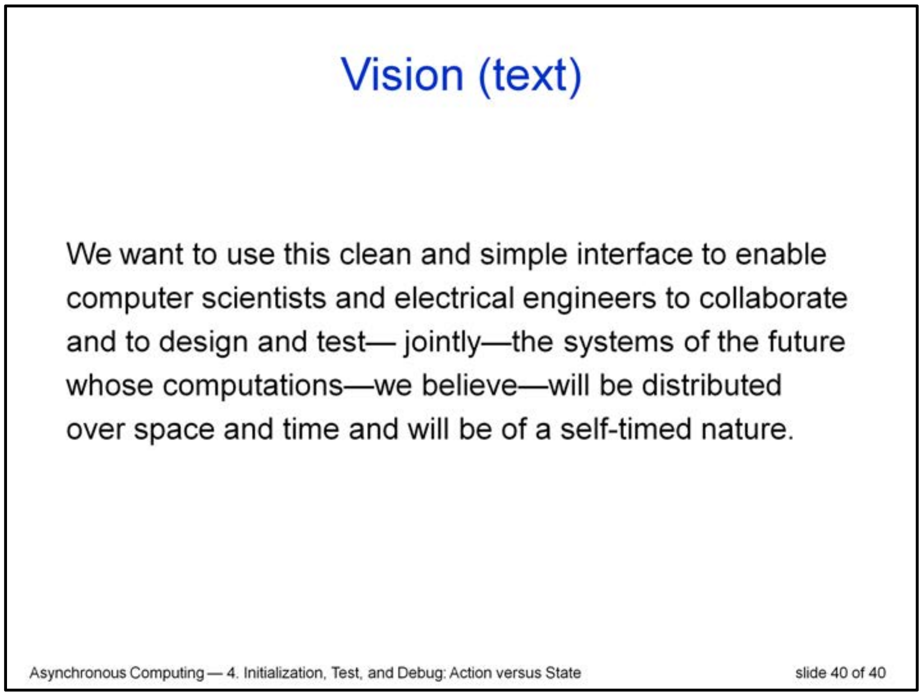
After: Kees van Berkel. Handshake Circuits, Fig. 1.1. Cambridge University Press, 1993.

Asynchronous Computing — 4. Initialization, Test, and Debug: Action versus State slide 39 of 40

I'd like to end by reminding everyone of our vision.

The link-joint model
with its full-empty protocol
and with its local action-state control
provides a clean and simple interface
for hardware-software co-design-and-test.

We want to use this clean and simple interface to enable
computer scientists and electrical engineers to collaborate
and to design and test - jointly - the systems of the future
whose computations - we believe - will be distributed
over space and time and will be of a self-timed nature.



Vision (text)

We want to use this clean and simple interface to enable computer scientists and electrical engineers to collaborate and to design and test— jointly—the systems of the future whose computations—we believe—will be distributed over space and time and will be of a self-timed nature.