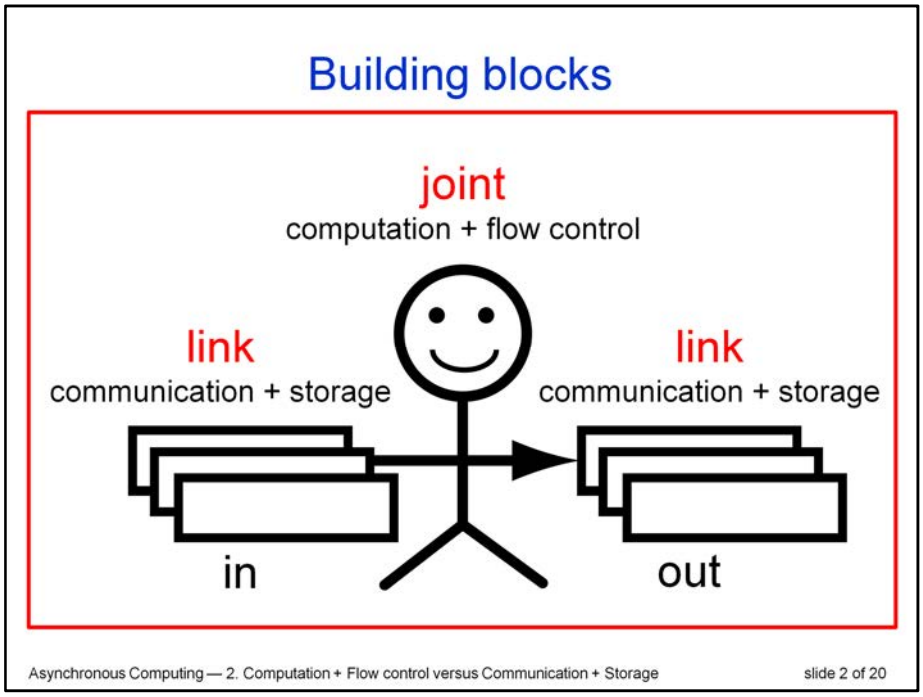
This is part 2 of our ShanghaiTech Lecture on Asynchronous Computing.

We will show how we separate - from the ground up -
communication and storage from computation and flow control,
and how we make them equal partners in asynchronous computing.



Here are the icons for our building blocks.

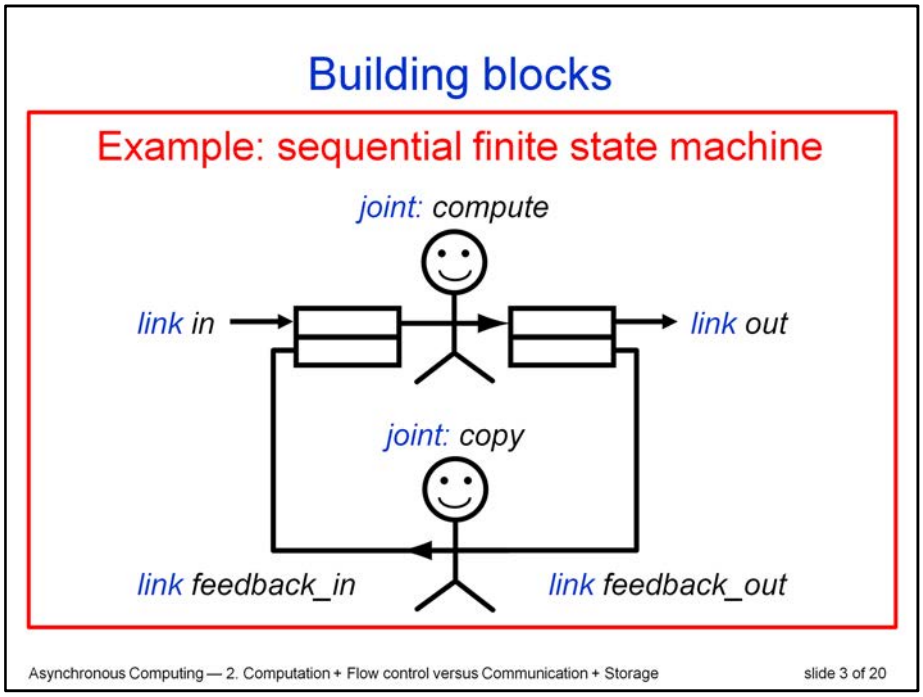
The stick-figure in the middle is a computing element.
We call it a ***joint***.
It works on data, but can also do flow control.
The data flow from left to right, in the direction of the arrow.
Data come in from the left, and computed results go out on the right.

The two boxes left and right of the stick-figure
bring the data to and from the computation.
We call each box a ***link***.
The box on the left carries the input data.
The box on the right carries the output results.

A joint may have zero or more input links
and zero or more output links.
<CLICK>

A link connects at most two joints,
one to give data **to** it
and the other to take data **from** it.

Links and joints alternate.

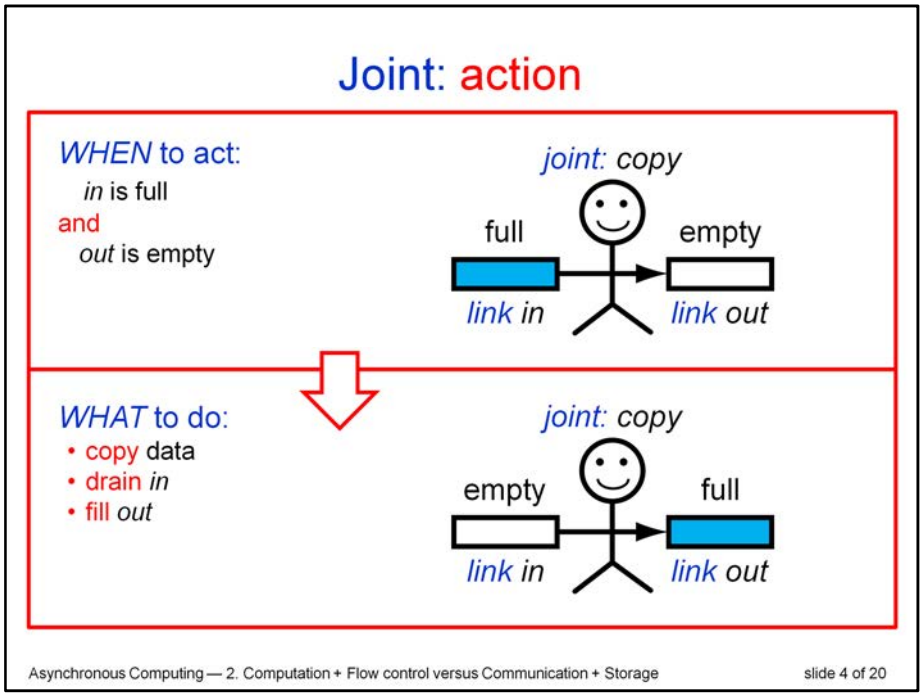


The computer scientists in the audience will appreciate that
this link-joint model can create a sequential finite state machine
<CLICK>
by storing the next internal state in a separate feedback link,
labeled feedback_out
<CLICK>
which is then copied by a separate joint
<CLICK>
onto another feedback link,
labeled feedback_in
to become the internal state input for the next computation step.

The copy operation in the feedback path is necessary
because a link NEVER reads AND writes data at the same time.

Note:
This will become clear when we explain the protocol for links and joints,
which we do in the next couple of slides.

We can do much more than sequential state machines.
The whole point of links and joints is to do parallel computing
distributed over space and time.



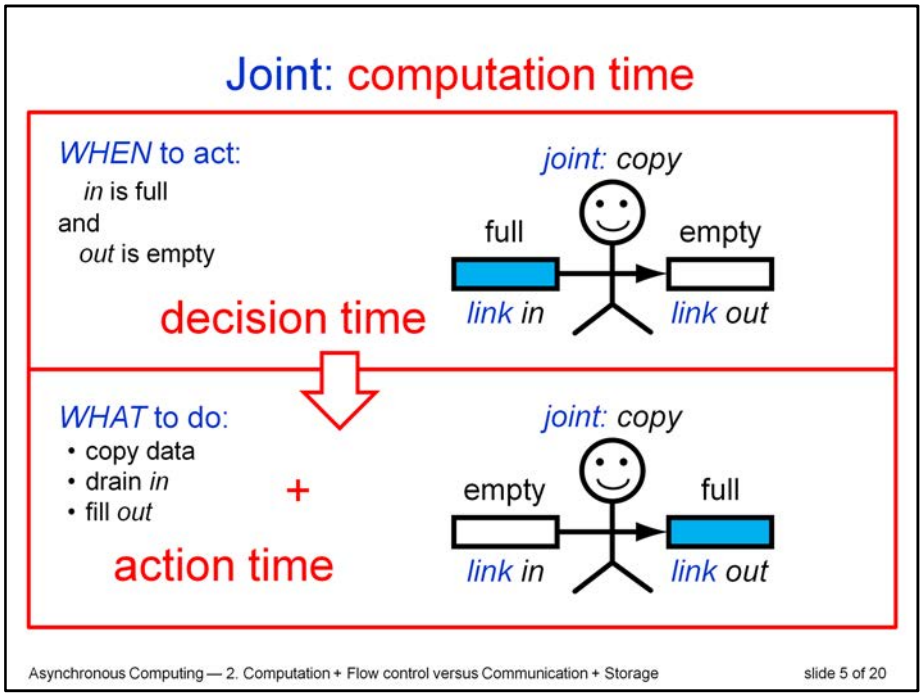
Here is an example of a simple joint
that copies data from its input link *in* to its output link *out*.
The joint can either do nothing or act.

The joint acts **when** link *in* is full and link *out* is empty.
Like this:
<CLICK>
We use blue for full and white for empty.
Full means that the data stored in the link are valid.
Empty means that the data stored in the link are irrelevant and can be replaced.

When the joint acts it
copies the data, drains link *in*, and fills link *out*
like this:
<CLICK>
Link *in* is now empty
and link *out* is now full.

By making link in empty and link out full,
the ***action enables other actions*** in neighboring joints.
But ***it disables itself!***

It is this self-disabling that makes asynchronous computing self-timed.
It's what makes it "tick" - without a clock.



I mentioned earlier that we want to compute over space and over time.

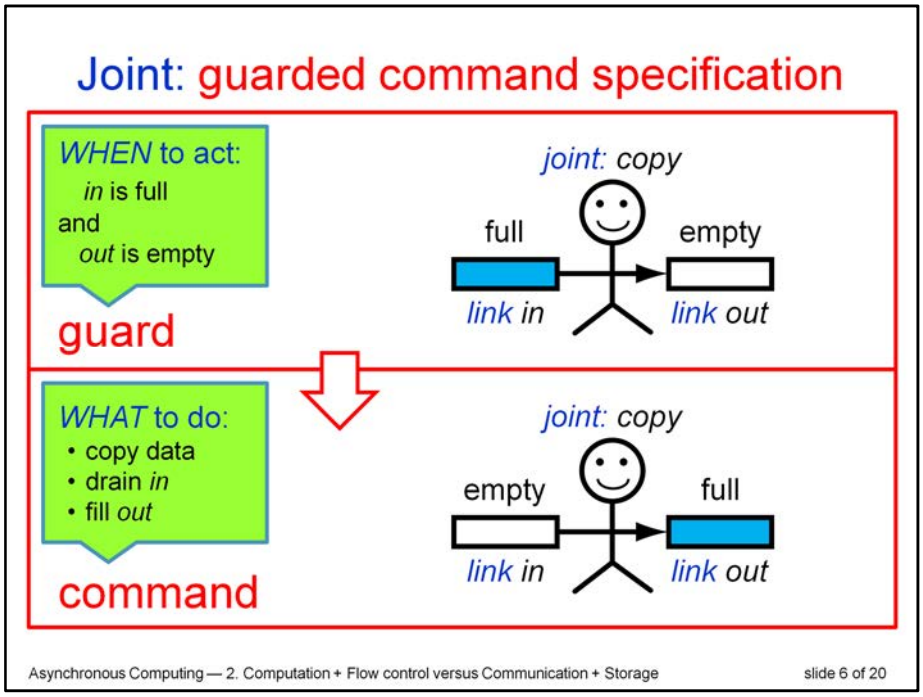
By separating communication and storage in links
and computation and flow control in joints,
we distribute actions over space - right from the start.

But what about time?
How is time distributed.

Links and joints act when they can.
Each action is local and can take a variable amount of time.

As example:
the computation for this joint starts when *in* is full and *out* is empty.

The time taken by the computation is a function of
<CLICK>
(1) the **decision** time to take this action
and
<CLICK>
(2) how long it takes the joint to **complete** this action
- in this case: how long it takes to copy, fill, and drain.

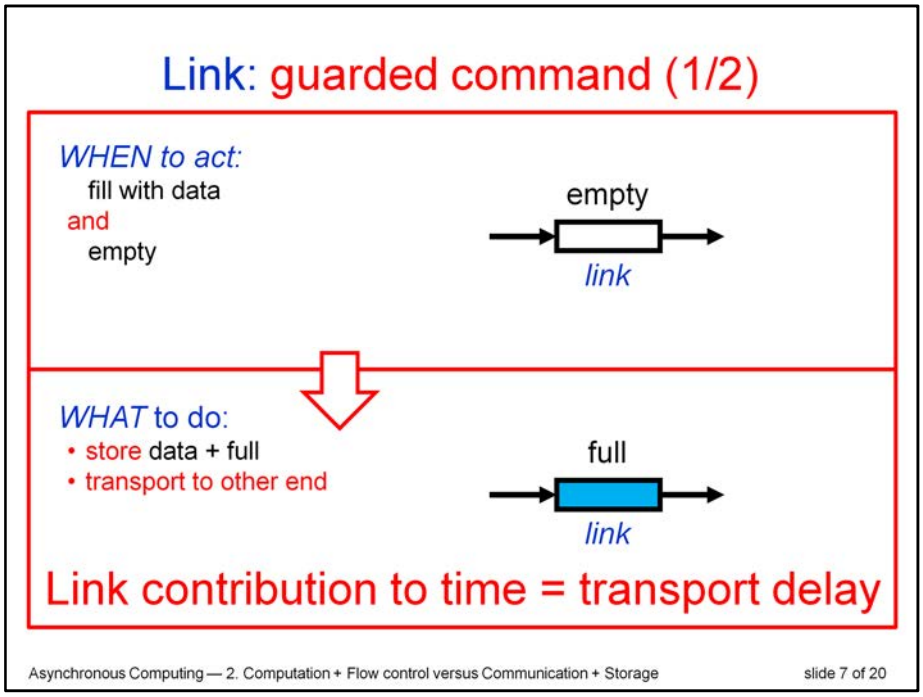


For those of you who are into formal methods,
it may be interesting to know that we use **Edsger Dijkstra's guarded commands** to specify what this joint can do.

Our **guarded command** notation is very similar to what's on this slide.

- The *when part* is the **guard**.
- <CLICK>
- The *what part* is the **command**.
- <CLICK>

The decision which guarded command to pick is deterministic in this case,
because this joint has exactly one guarded command.
We will discuss non-deterministic joints later.



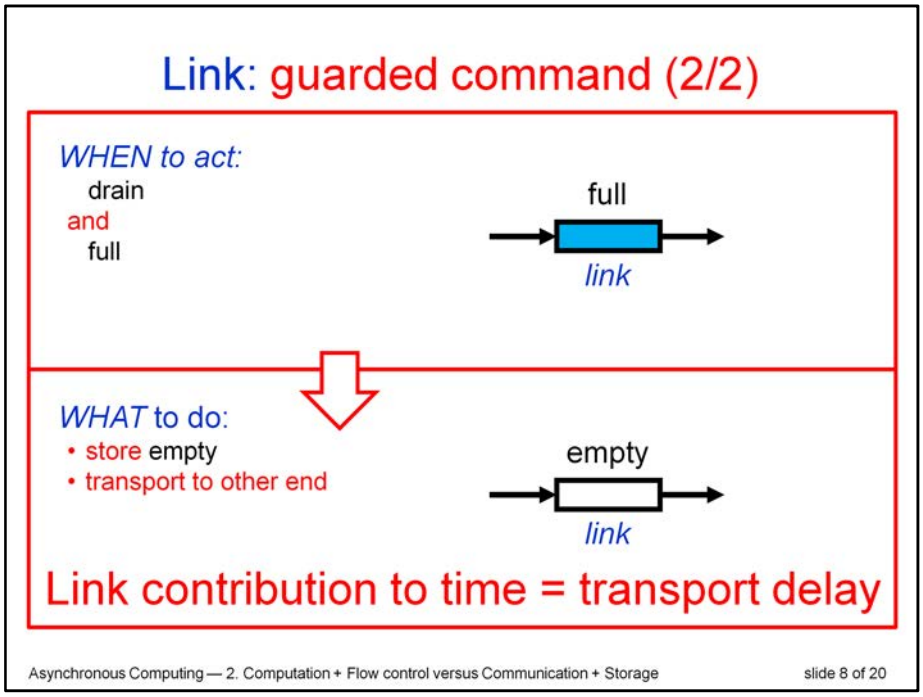
What about links?
What's the protocol for links?

Well, as you probably already expect,
a link has **TWO** guarded commands.
One to fill an empty link, making it full,
and another one to drain a full link, making it empty.

When an empty link is filled.
it **stores the data** that it receives
it **stores its new full state** information,
and then it **transports both** to its other end - like this:
<CLICK>

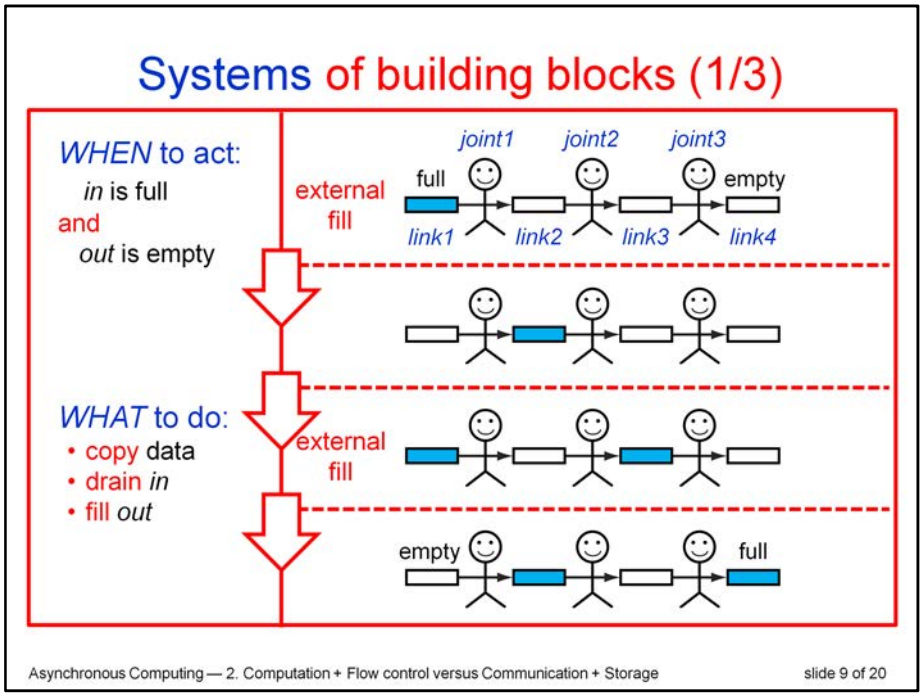
Storing the link's data and full state information
is an atomic action with the **filling** joint.

The link's contribution to time is its transport delay.



Likewise:
When a full link is drained.
it **stores its new empty** state information,
and **transports it** to the other end, like this:
<CLICK>

Storing the link's empty state information
is an atomic action with the **draining** joint.
The link's main contribution to time is its transport delay.



We can build a First In First Out buffer, or FIFO,
by connecting multiple links and joints in series,
as in this picture.

This FIFO has three joints: joint1-2-3.
They're instances of the joint you saw a few slides ago.
They copy data from their input link to their output link,
whenever the input link is full and the output link is empty.

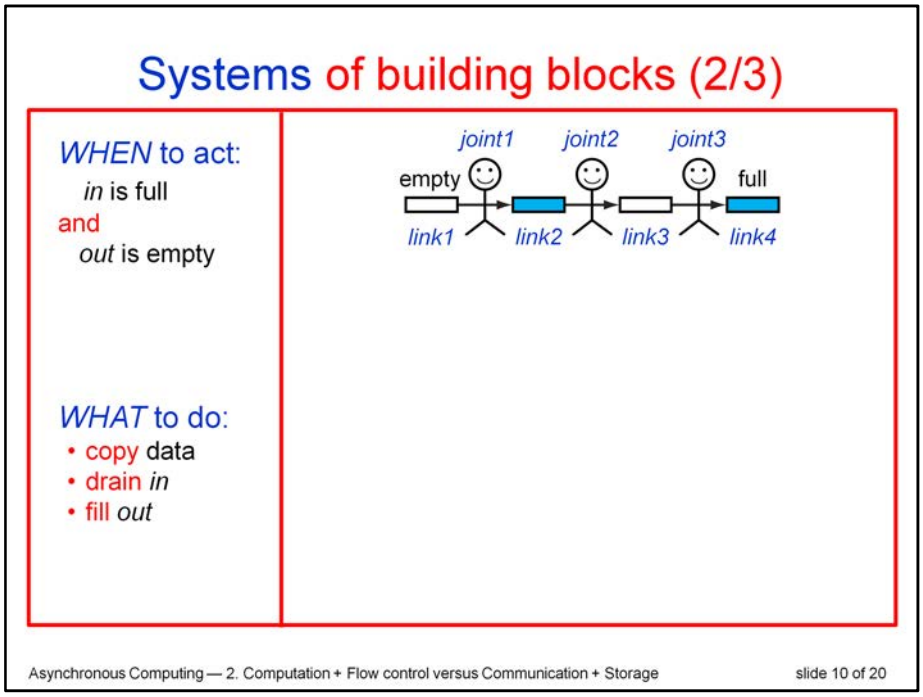
As a reminder, the left-side of this slide shows
the guarded command specification for each joint.

The FIFO has four links: link1-2-3-4.
Given that the links store the data,
this FIFO can store up to 4 data items.

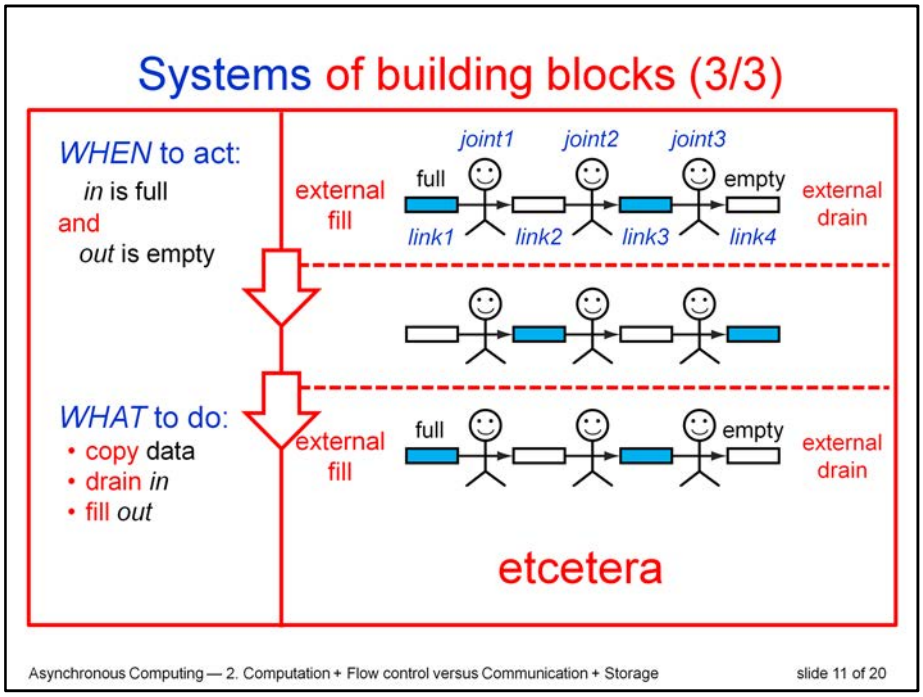
Initially, this FIFO has no relevant data, because all its links are empty.
Let's fill link1 every time it's empty
and let's drain link4 every time it's full,
and see what happens.

<CLICK - ANIMATION STOPS AT ROW 3>
Notice that **joint1** and **joint3** can now operate in parallel,
because their input links are full and their output links are empty.
As a result, all links will change state in parallel.
<CLICK TO GET ROW 4>

This is no longer a sequential finite state machine.
This is a parallel machine.



This is the end state of the previous slide.
Note that **joint2** can act in parallel with the external fill and drain environment.
As a result, all four links will change state in parallel.



<STARTS AT ROW1 with empty-full-empty-full>
<CONTINUES AUTOMATICALLY to full-empty-full-empty>

And again - all four links can change state in parallel.

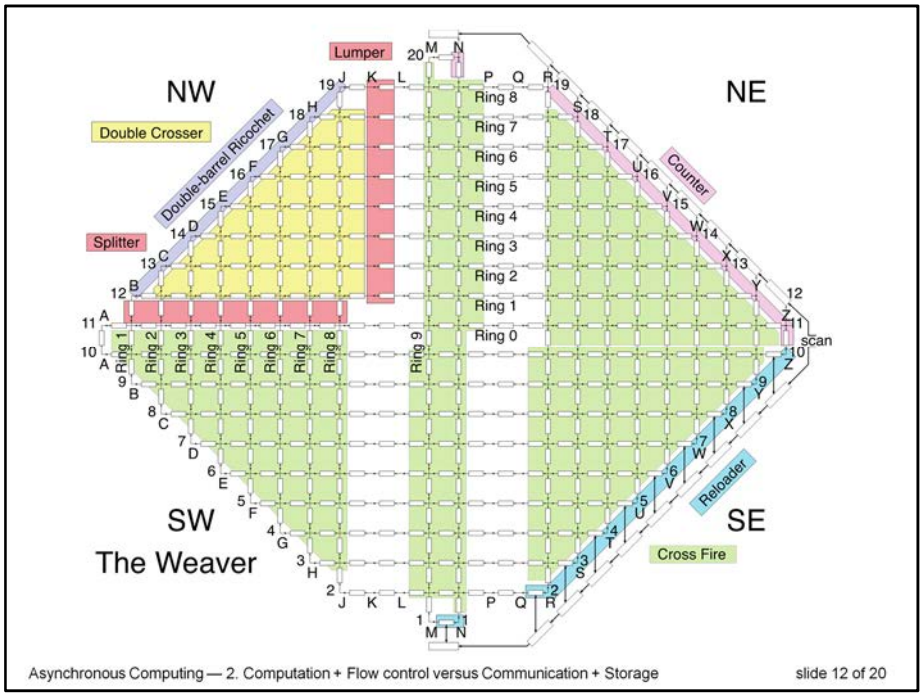
<CLICK to end at ETCETERA>

Note that we're back in the same full-empty state as in the topmost FIFO;
only the data are different.

If we sustain the filling and draining, then
this FIFO will be operating at a maximum number of data items per cycle.

In other words, this slide gives you a snapshot
of the FIFO operating at its maximum throughput.

Keep that thought...
because in the next two slides
we will show how you measure and represent throughput
of an asynchronous self-timed system.



FIFOs play a big role in on-chip communication.

Here is a link-joint representation of our Weaver chip,
an 8x8 crossbar, built in 2015 in 40nm CMOS by TSMC.

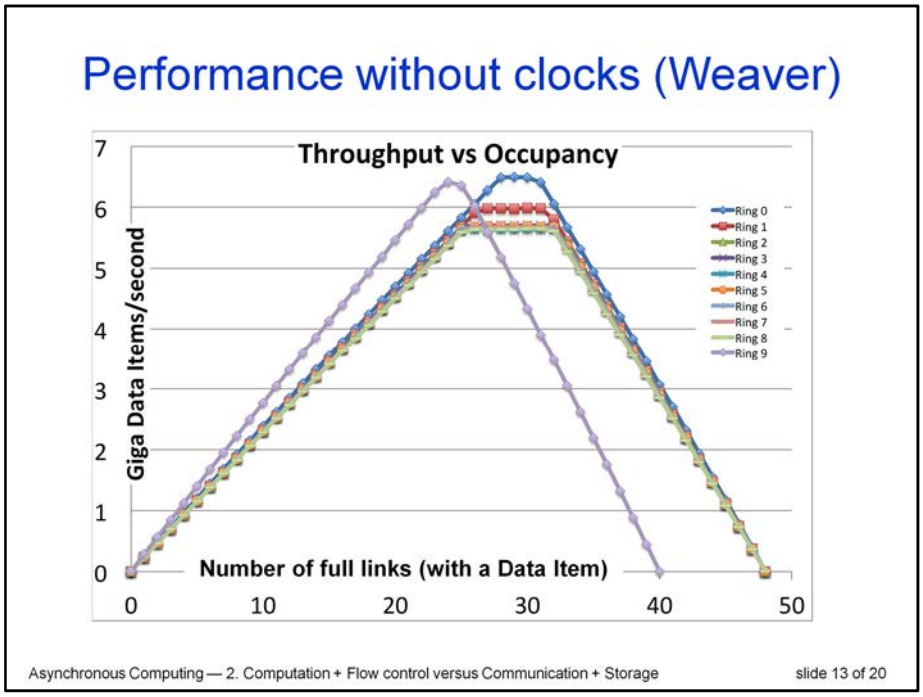
Rectangles are links.
Dots are joints.

The yellow area in the NW corner is the crossbar.

There are 10 ring FIFOs with 40 to 48 links each.
They go around like a folded ribbon.
Eight go through the crossbar where they can pass data to any of the other eight.
Data circulate clockwise.

A scan interface and re-loaders in the SE corner let us initialize and test the chip.

Ring counters in the NE corner increment each time a data item zips by.
We use these counters to measure the throughput for each ring in the Weaver.
We measure the throughput by
checking how many data items are counted per second.



Here are the throughput results.

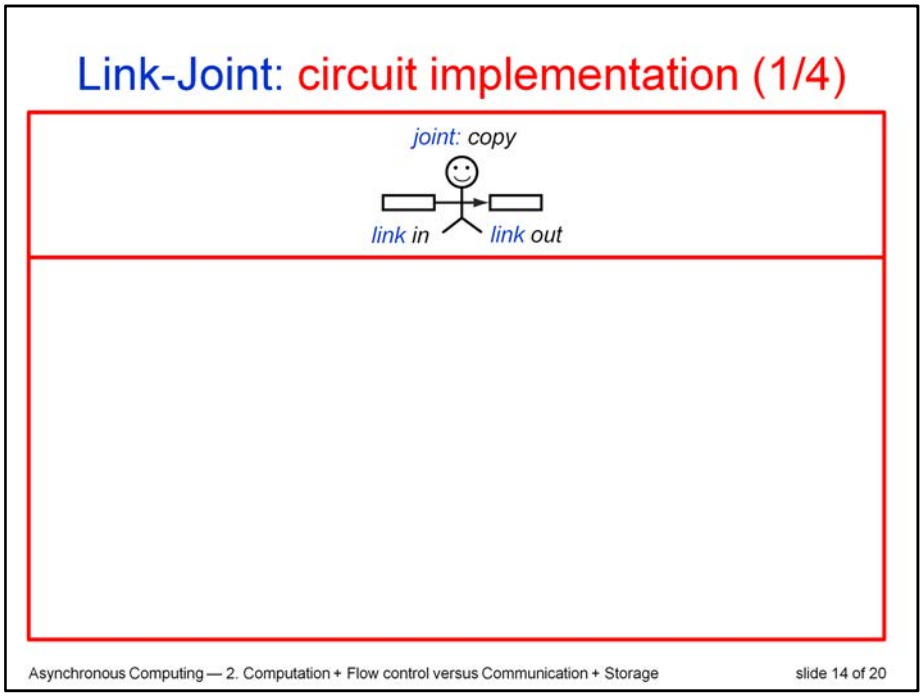
The horizontal axis shows the number of full links per ring.
The vertical axis shows the throughput of each ring
measured in Giga Data Items per second.
The graphs show the throughput for the various rings, for various occupancies.

The throughput is lowest
when the FIFO is completely full with data or completely empty.
With two full links, the throughput is twice as high as with one full link.
With two empty links the throughput is twice as high as with one empty link.
The throughput is highest when the FIFO is 60% full.
This gives the graphs a tent-like shape.
This is why these graphs are known as "canopy graphs."
A canopy is a tent-like roof structure.

In the Weaver, spaces move faster than data - empty moves faster than full.
This explains why the right-hand slopes of the graphs are steeper
than the left-hand slopes, and why the highest throughput occurs
not when 50% of the links are full ***but*** when 60% of the links in the ring are full.

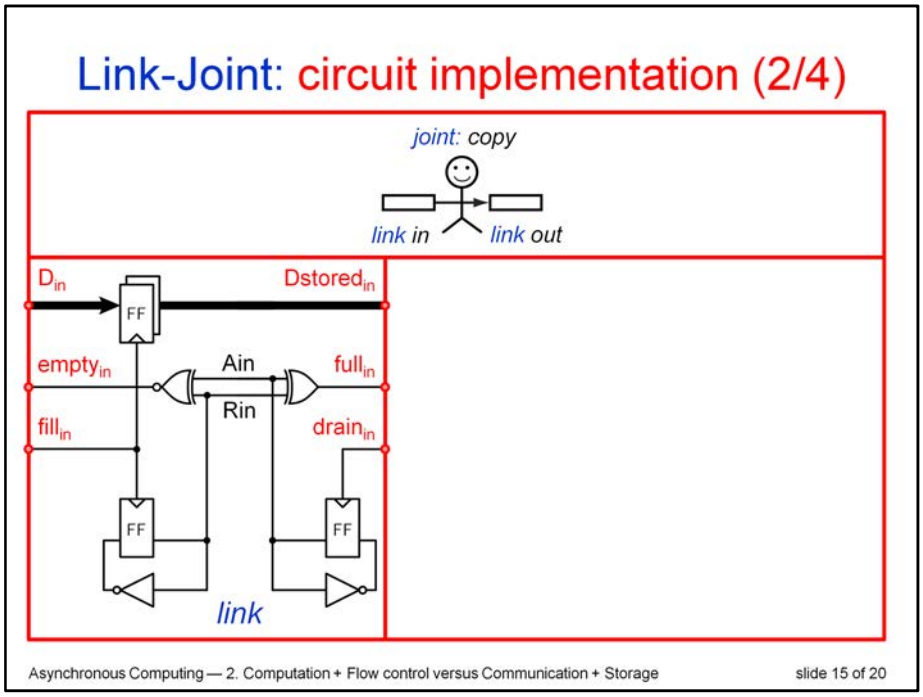
I show this slide for two reasons

- One, to show how to measure performance without clocks.
- Two, to emphasize how fast link and joint designs are.
  Each ring runs at about 6 GigaDataItems per second
  - much faster (~2x) than a globally clocked design.
  For 8 crossbar rings with 72 bit-wide data,
  that's 3.5 Terabits per second (Tb/s) for the full crossbar.



Links and joints have a circuit implementation,
which behaves exactly as their guarded command specifications.

As example, let's look at the circuit implementation of a basic FIFO stage
that copies data from link *in* to link *out*.



The input link can be implemented as follows.
You can see the interfaces:

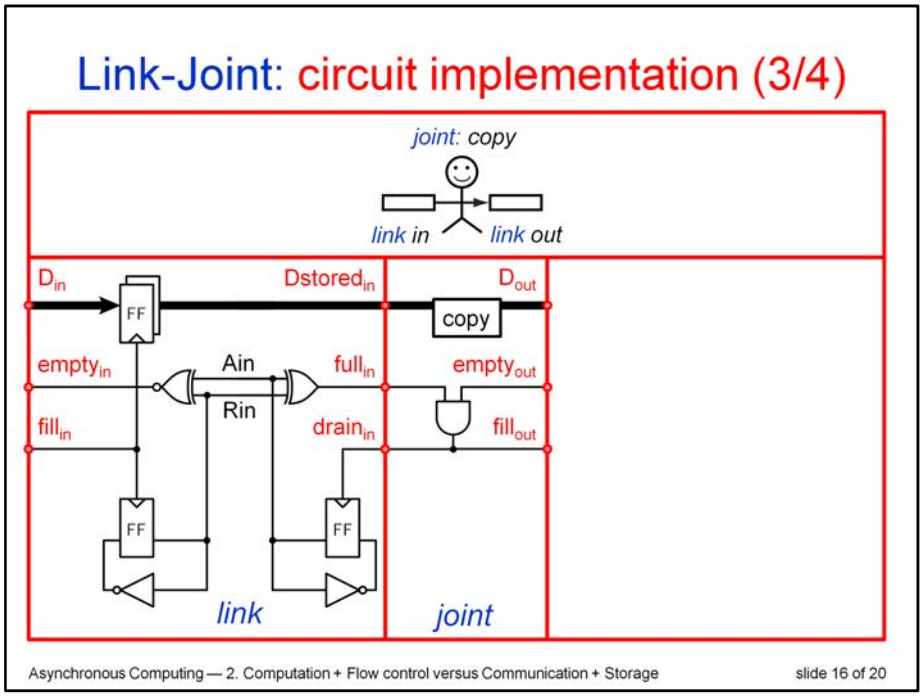
- full, empty, fill, drain, and data

Data go from left to right.

The flipflops at the top store the incoming data.
The two flipflops at the bottom **together** store the full-or-empty state.

Fill comes in on the left,
makes an empty link full,
and transports that full state information
to the other end - on the right.

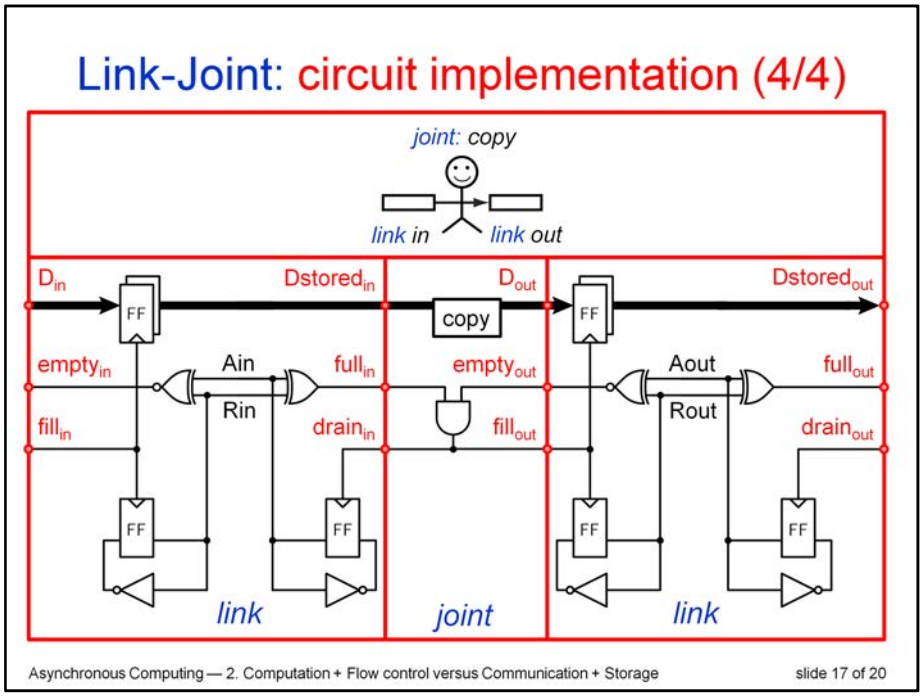
Drain comes in on the right,
makes a full link empty,
and transports that empty state information
to the other end - on the left.



The joint can be implemented as a simple AND gate
for ANDing a full in with an empty out.

If the AND goes high,
the joint drains link *in*
and fills link *out*.

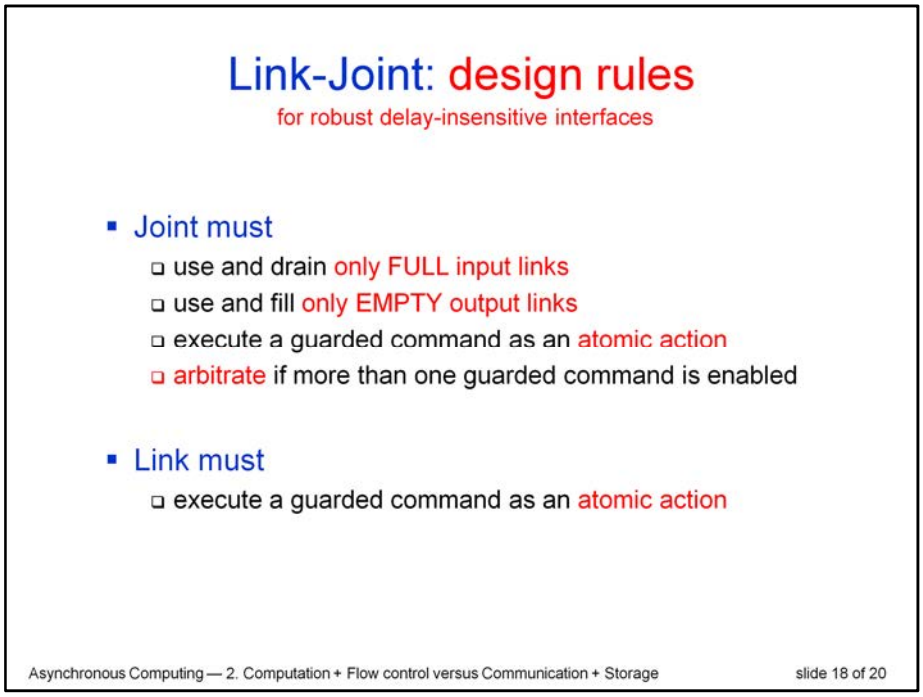
The joint also has combinational logic
to connect and copy the data from link *in* to link *out*.



The output link can have the same implementation as the input link.

It could also be implemented with a different circuit
that uses a different handshake protocol.

The good news about the link-joint model is that it unifies
existing asynchronous self-timed circuit families.
You can use whichever family fits best for your application.
If you need speed, use GasP - as we did in the Weaver.
If you want to use standard industry tools, use Click - as we do on this slide.



You can make any joint you like,
one that copies data as I showed earlier,
one that adds or sorts,
one that merges data streams,
or selectively connects data streams,
ANYTHING you like.

BUT:
to keep the connection to circuit implementations with robust interfaces
we have the following design rules:

A joint must
- Use and drain only FULL input links
- This also means that the data it computes on
  must be from full input links

A joint must
- Use and fill only EMPTY output links

It must
- Execute a guarded command as an **atomic action**

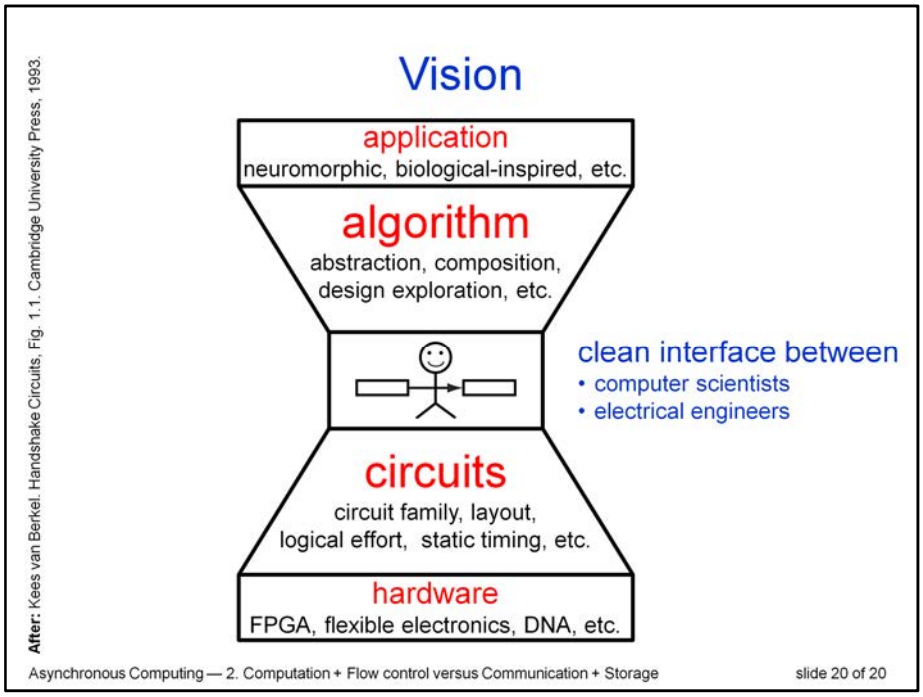
And if more than one guarded command is enabled, it must
- pick one **using arbitration**

Ivan will discuss arbitration in the next talk.

A link must also
- Execute a guarded command **as an atomic action**

Summary

- Communication and computation are equal partners
  - from the bottom up
  - links do communication and storage
  - joints do computation and flow control
- Interfaces matter
  - design them for collaboration and re-use
  - full-empty interface
    - works for computer scientists and electrical engineers
    - unifies self-timed circuit families



Why do we do this?
We do this because it fits our bigger vision.
The link-joint model with its full-empty protocol is simple enough
so that computer scientists and electrical engineers can **both** work with it.

Starting at the link-joint model and above are the algorithms.
<CLICK>
The questions and answers here relate to design exploration
for domain-specific applications.

Starting at the link-joint model and below are the circuit implementations.
<CLICK>
The questions and answers here relate to electrical integrity
and technology mapping.

The questions and answers above and below the link-joint model
are sufficiently different to want to shield
the peculiar details from each other.

Links and joints make a good shield and a good hardware-software interface.
They're the new RTL (Register Transfer Level)
for space-and-time distributed computing.
<CLICK>
Many upcoming systems are distributed over space and time.
I mention a few on this slide:

- neuromorphic computing (see top)
- and DNA-RNA computing (see bottom)
- both of which - by the way - are of a **self-timed nature !**