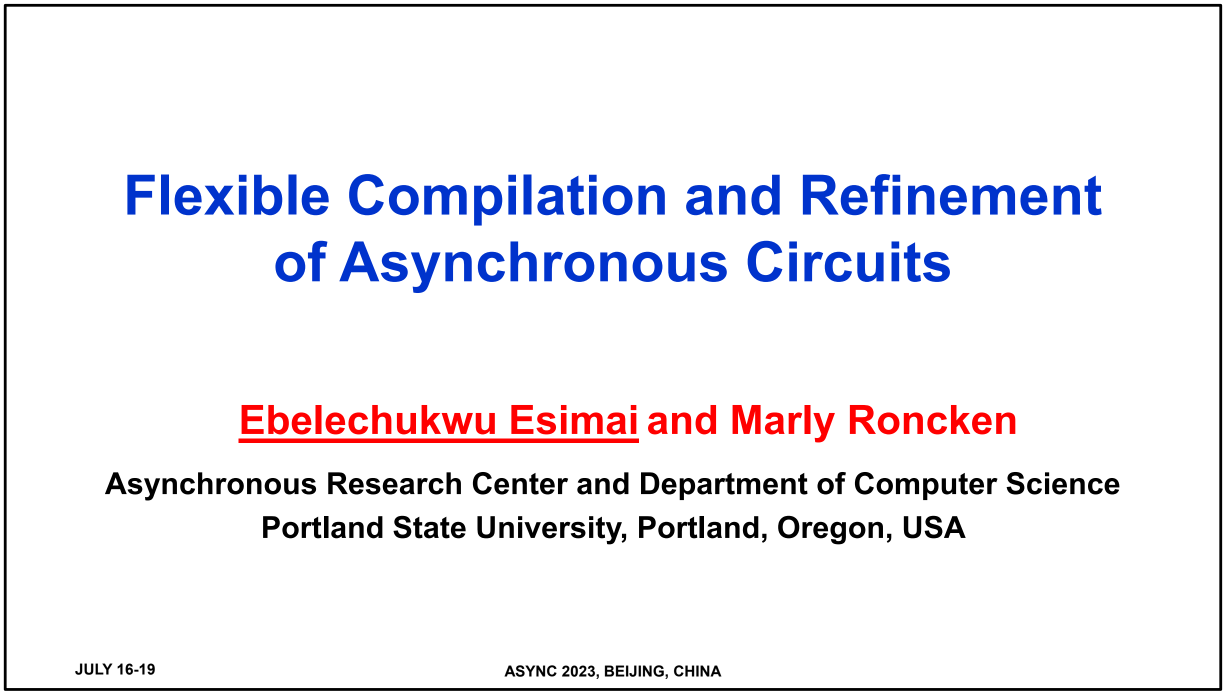


Flexible Compilation and Refinement of Asynchronous Circuits

Ebelechukwu Esimai and Marly Roncken

Asynchronous Research Center and Department of Computer Science
Portland State University, Portland, Oregon, USA

JULY 16-19

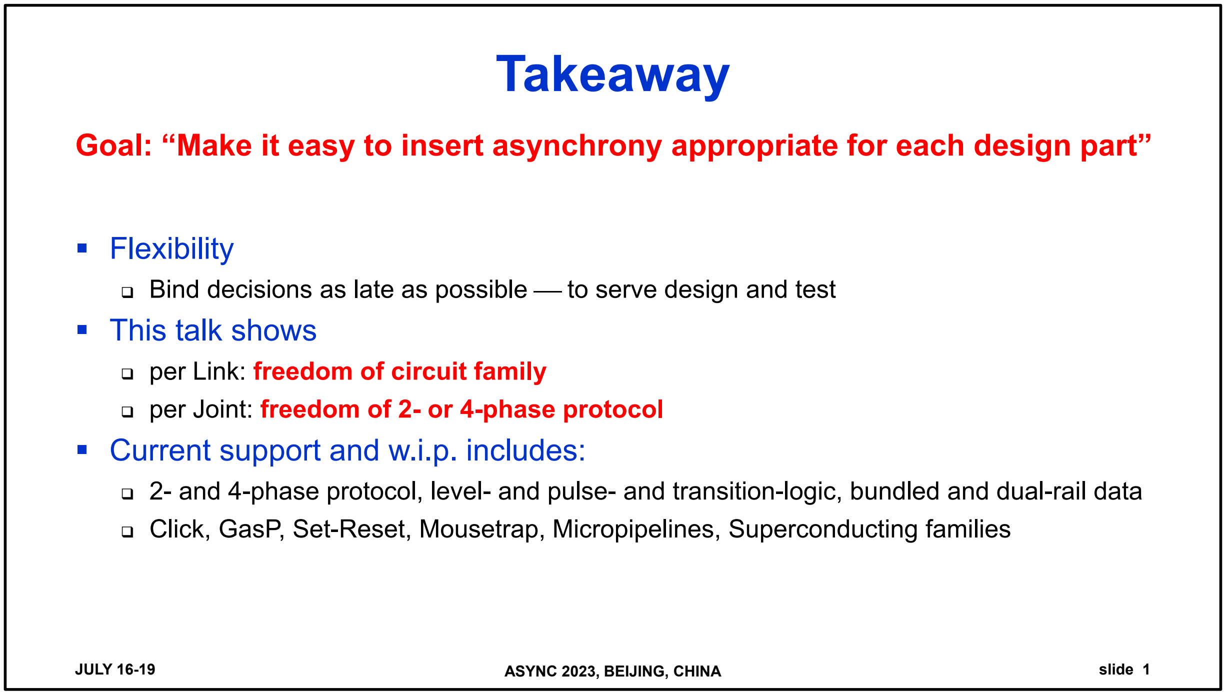
ASYNC 2023, BEIJING, CHINA

“Hello” and “Ni Hao” (pronounce: “knee how”) !

My name is Ebele.
I work with Marly Roncken in the Asynchronous Research Center
at Portland State University.

I am honored to be here in China to present our paper titled

- Flexible Compilation and Refinement of Asynchronous Circuits



Takeaway

Goal: “Make it easy to insert asynchrony appropriate for each design part”

- Flexibility
  - Bind decisions as late as possible — to serve design and test
- This talk shows
  - per Link: **freedom of circuit family**
  - per Joint: **freedom of 2- or 4-phase protocol**
- Current support and w.i.p. includes:
  - 2- and 4-phase protocol, level- and pulse- and transition-logic, bundled and dual-rail data
  - Click, GasP, Set-Reset, Mousetrap, Micropipelines, Superconducting families

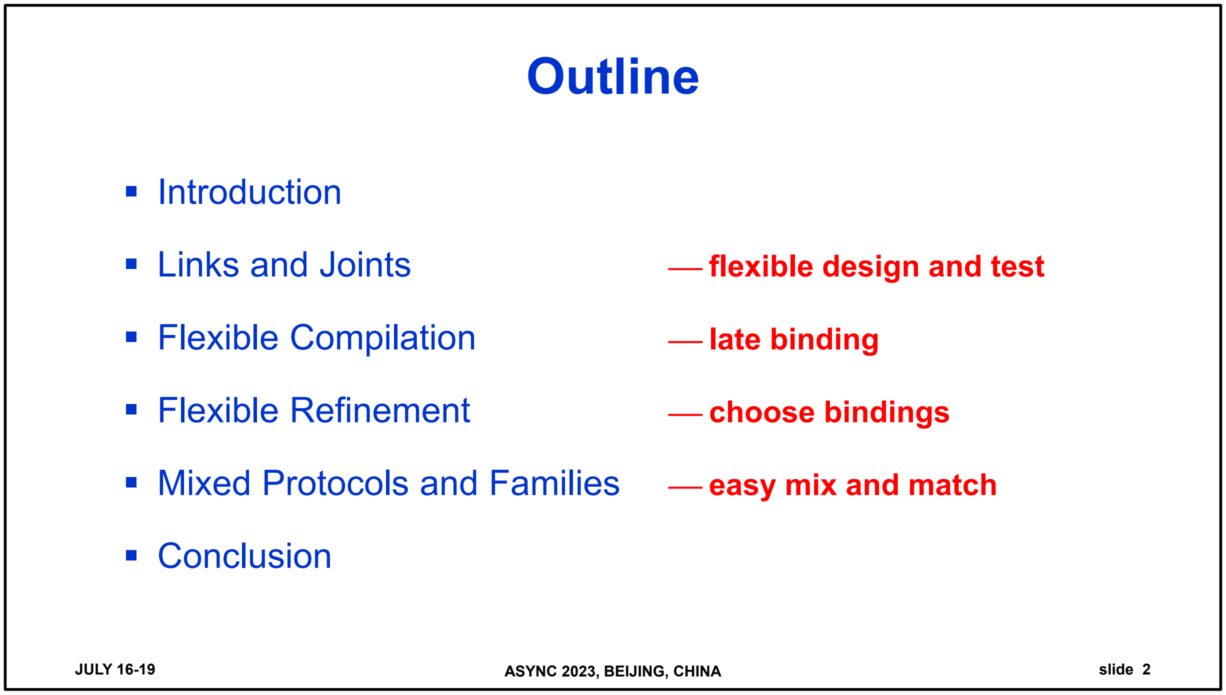
Before I go into the details of the paper, here is what to takeaway from our approach.
When one is designing asynchronous circuits, there are many decisions to make,
Such as communication protocol, data encoding and circuit family.
One may base some of these decisions on one’s familiarity and experience.

Rather, with our approach, Instead of making these decisions early, we bind decisions as late as possible.
This makes our approach flexible.
For example:

- Each Link can choose its circuit family freely, independent of other Links.
- Each Joint can choose freely to use a 2-phase or a 4-phase protocol, independent of other Joints.

Here is a list of protocols, logic, data encodings and circuit families that we support or are working on.

The goal is to Make it easy to insert asynchrony appropriate to each design part!



Outline

- Introduction
- Links and Joints — flexible design and test
- Flexible Compilation — late binding
- Flexible Refinement — choose bindings
- Mixed Protocols and Families — easy mix and match
- Conclusion

Here is the outline of my talk.

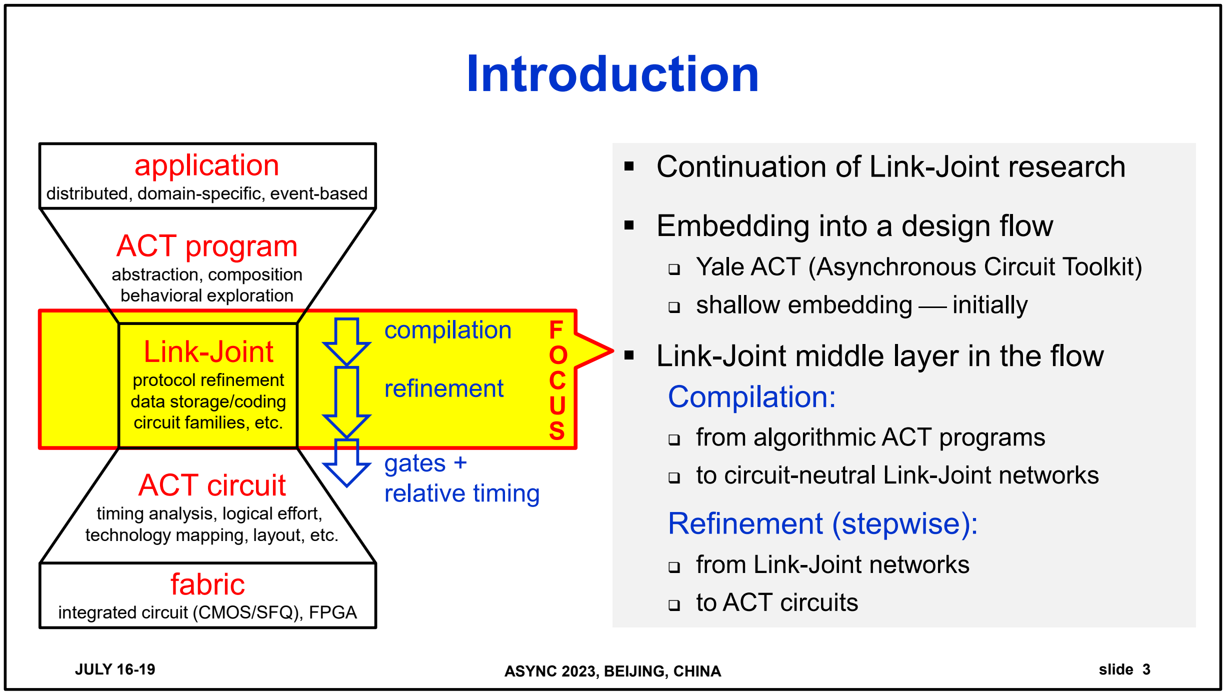
I will start with an introduction and a brief summary

- of the way we design asynchronous circuits using Links and Joints - for flexible design and test

I will then move to the key parts:

- flexible compilation (for late binding)
- and flexible refinement (to explore and choose bindings)

I will end by giving an example that shows that mixing and matching
Different protocols and circuit families is easy with our approach.



Introduction

application

distributed, domain-specific, event-based

ACT program

abstraction, composition
behavioral exploration

Link-Joint

protocol refinement
data storage/coding
circuit families, etc.

ACT circuit

timing analysis, logical effort,
technology mapping, layout, etc.

fabric

integrated circuit (CMOS/SFQ), FPGA

compilation

refinement

gates +
relative timing

FOCUS

- Continuation of Link-Joint research
- Embedding into a design flow
  - Yale ACT (Asynchronous Circuit Toolkit)
  - shallow embedding — initially
- Link-Joint middle layer in the flow

  Compilation:
  - from algorithmic ACT programs
  - to circuit-neutral Link-Joint networks

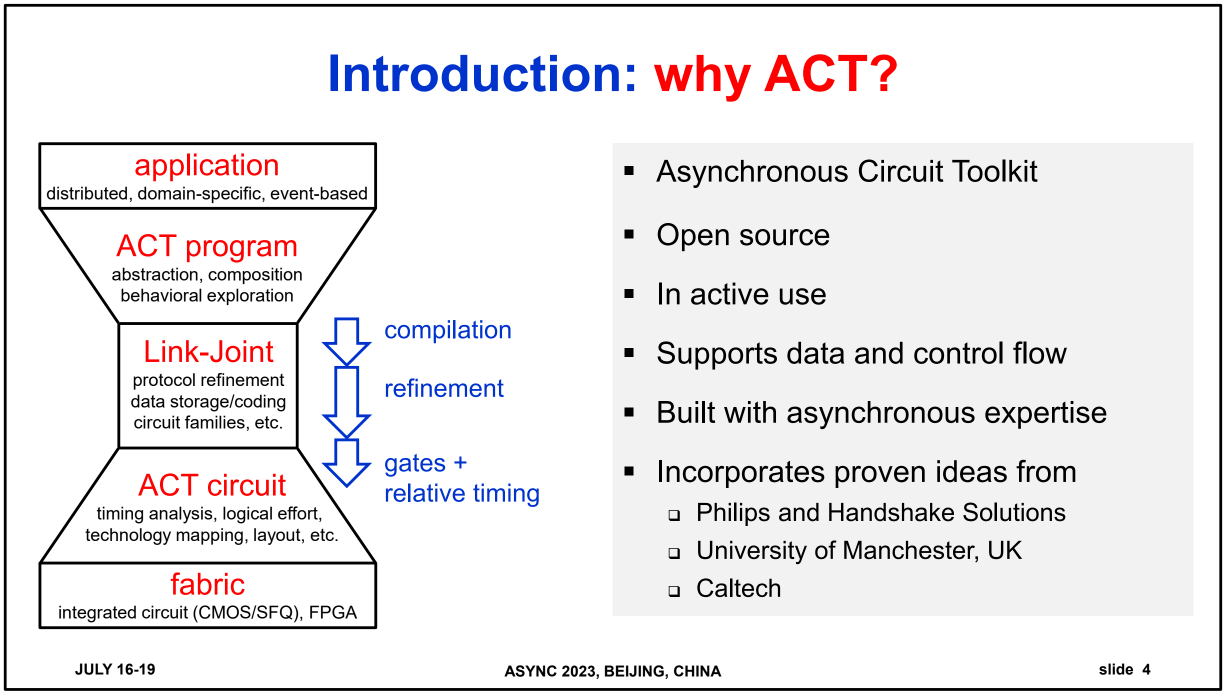
  Refinement (stepwise):
  - from Link-Joint networks
  - to ACT circuits

This work continues the Link-Joint research done at Portland State University by embedding Links and Joints into a design flow.

As our design flow, We use Yale's Asynchronous Circuit toolkit – ACT,
as introduced in the tutorial session today by Rajit Manohar.
We depict this flow with the hourglass on the left.

We compile ACT programs into **circuit-neutral** Link-Joint networks,
which we then translate into ACT circuits, using stepwise refinement.

The key focus of this talk is on compilation and refinement.



Introduction: why ACT?

application
distributed, domain-specific, event-based

ACT program
abstraction, composition
behavioral exploration

Link-Joint
protocol refinement
data storage/coding
circuit families, etc.

ACT circuit
timing analysis, logical effort,
technology mapping, layout, etc.

fabric
integrated circuit (CMOS/SFQ), FPGA

compilation

refinement

gates +
relative timing

- Asynchronous Circuit Toolkit
- Open source
- In active use
- Supports data and control flow
- Built with asynchronous expertise
- Incorporates proven ideas from
  - ❑ Philips and Handshake Solutions
  - ❑ University of Manchester, UK
  - ❑ Caltech

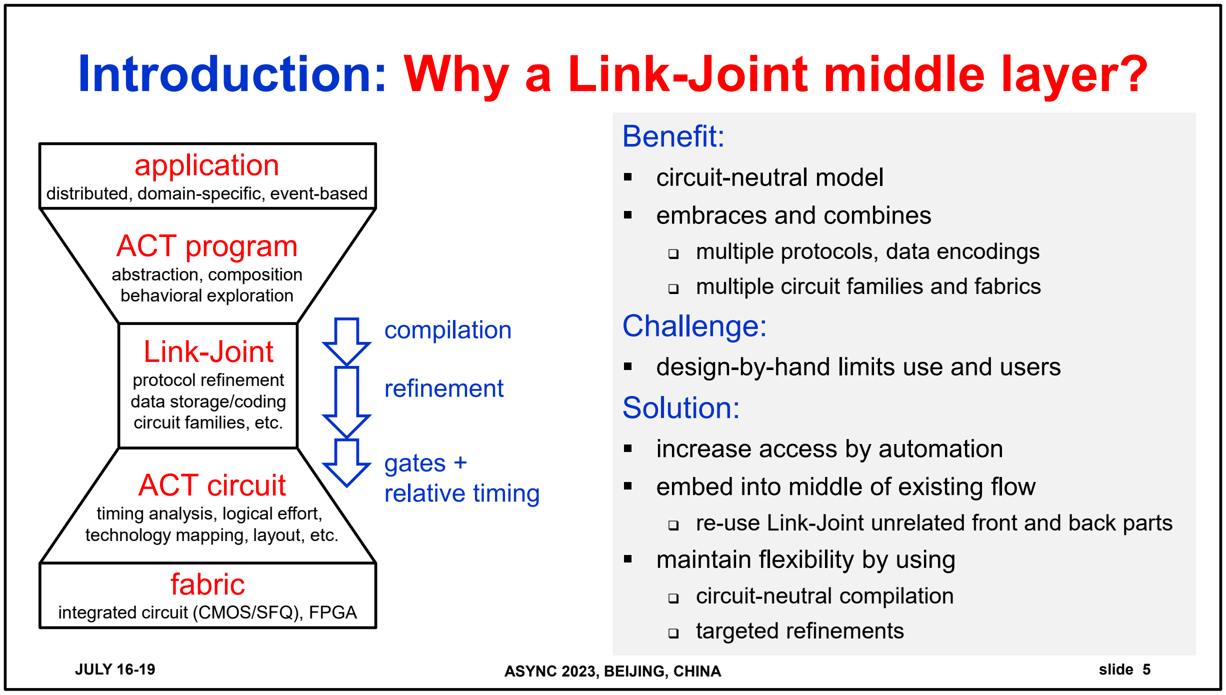
We use ACT as our electronic design automation flow,
because

- it is open-source, in active use
- and supports programs with data- as well as control-flow.

Moreover, ACT is built with in-depth knowledge

- about asynchronous design and analysis

And incorporates proven ideas and solutions from Philips, Manchester and Caltech.
<Pause 2>



Introduction: Why a Link-Joint middle layer?

application
distributed, domain-specific, event-based

ACT program
abstraction, composition
behavioral exploration

Link-Joint
protocol refinement
data storage/coding
circuit families, etc.

ACT circuit
timing analysis, logical effort,
technology mapping, layout, etc.

fabric
integrated circuit (CMOS/SFQ), FPGA

compilation

refinement

gates +
relative timing

Benefit:

- circuit-neutral model
- embraces and combines
  - multiple protocols, data encodings
  - multiple circuit families and fabrics

Challenge:

- design-by-hand limits use and users

Solution:

- increase access by automation
- embed into middle of existing flow
  - re-use Link-Joint unrelated front and back parts
- maintain flexibility by using
  - circuit-neutral compilation
  - targeted refinements

JULY 16-19 ASYNC 2023, BEIJING, CHINA slide 5

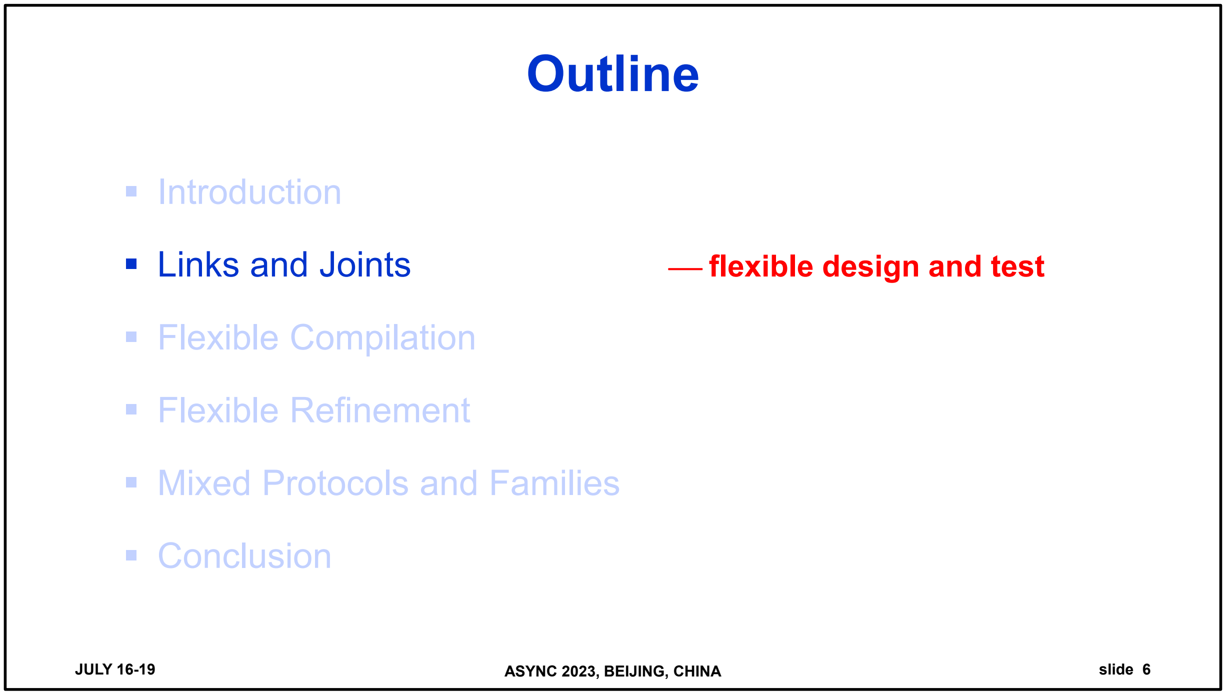
We use Links and Joints because they provide
- a circuit-neutral design and test model

that embraces multiple protocols, data encodings, families and fabrics.

In the past, we had to do Link-Joint designs by hand,
which limits both its use and the number of users.
By embedding Links and Joints into an automated flow,
we can expand its reach - do larger designs and have more users.

We re-use the design flow's
- application and programming techniques at the top
- and circuit and fabrication techniques at the bottom.

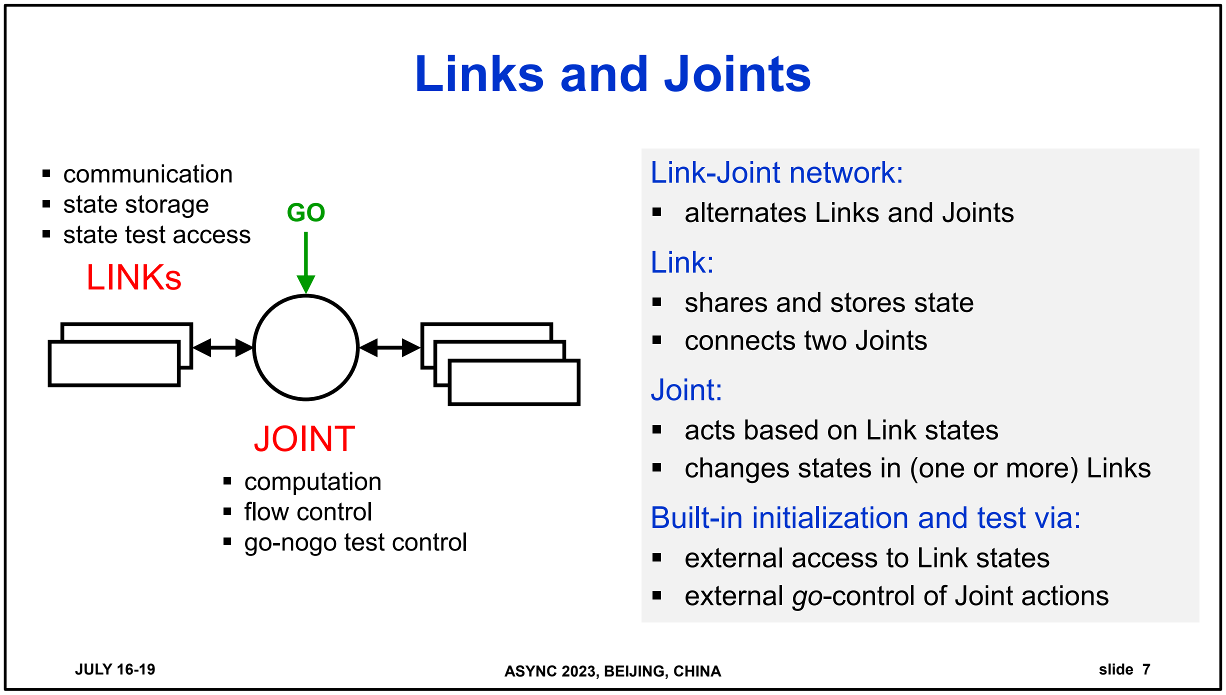
Therefore, we have Links and Joints in the middle of the flow.



Outline

- Introduction
- Links and Joints — flexible design and test
- Flexible Compilation
- Flexible Refinement
- Mixed Protocols and Families
- Conclusion

Now, I will give a brief summary of Links and Joints for flexible design and test.



Links and Joints

- communication
- state storage
- state test access

GO

LINKs

JOINT

- computation
- flow control
- go-nogo test control

Link-Joint network:

- alternates Links and Joints

Link:

- shares and stores state
- connects two Joints

Joint:

- acts based on Link states
- changes states in (one or more) Links

Built-in initialization and test via:

- external access to Link states
- external *go*-control of Joint actions

JULY 16-19 ASYNC 2023, BEIJING, CHINA slide 7

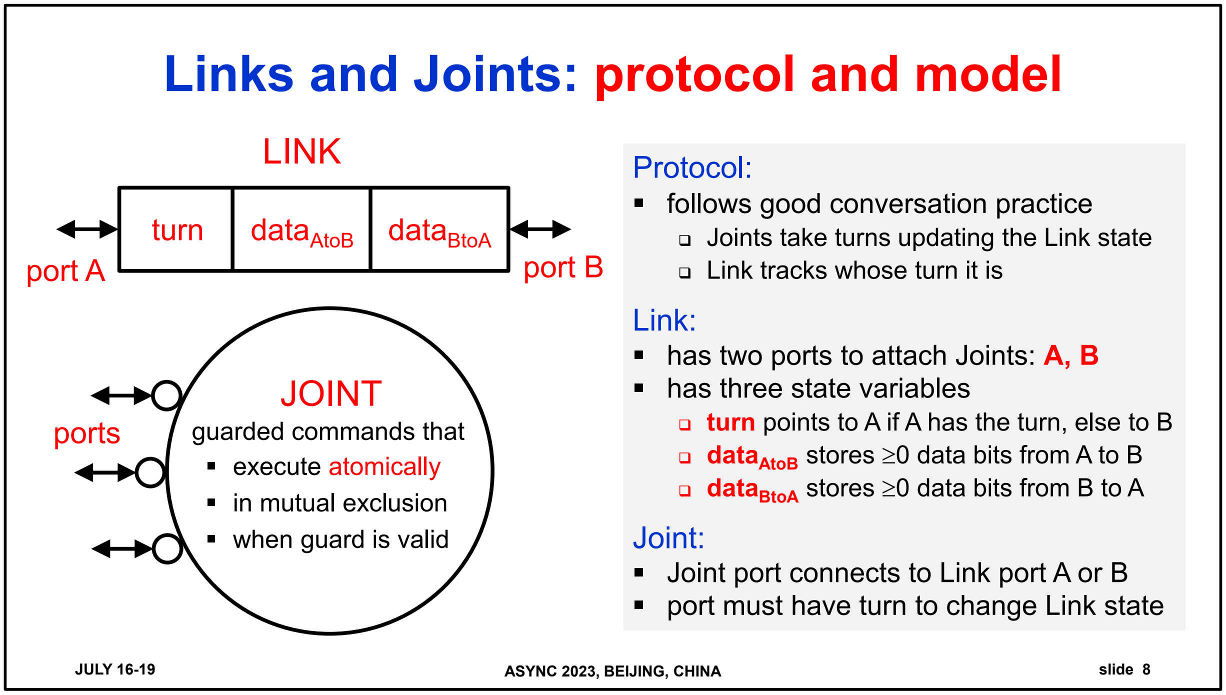
The picture on the left shows a Link-Joint network.
We draw Links as rectangles and Joints as circles. A Link-Joint network alternates Links and Joints.

A Link is a communication channel that stores state.
While a Joint is a computation module with flow control.

- A Joint acts on Link states and changes these.

Note that Links and Joints have built-in initialization and test through

- external access to Link states, and
- external *go*-control to enable or disable Joint actions.



Links and Joints: protocol and model

LINK

port A

turn

data$_{AtoB}$

data$_{BtoA}$

port B

ports

JOINT

guarded commands that
- execute atomically
- in mutual exclusion
- when guard is valid

Protocol:
- follows good conversation practice
  - Joints take turns updating the Link state
  - Link tracks whose turn it is

Link:
- has two ports to attach Joints: **A, B**
- has three state variables
  - **turn** points to A if A has the turn, else to B
  - **data$_{AtoB}$** stores ≥0 data bits from A to B
  - **data$_{BtoA}$** stores ≥0 data bits from B to A

Joint:
- Joint port connects to Link port A or B
- port must have turn to change Link state

The Link-Joint protocol follows good conversation practice where one listen while the other talks.
After a while, you switch roles and TAKE TURNS between who listens and who talks.

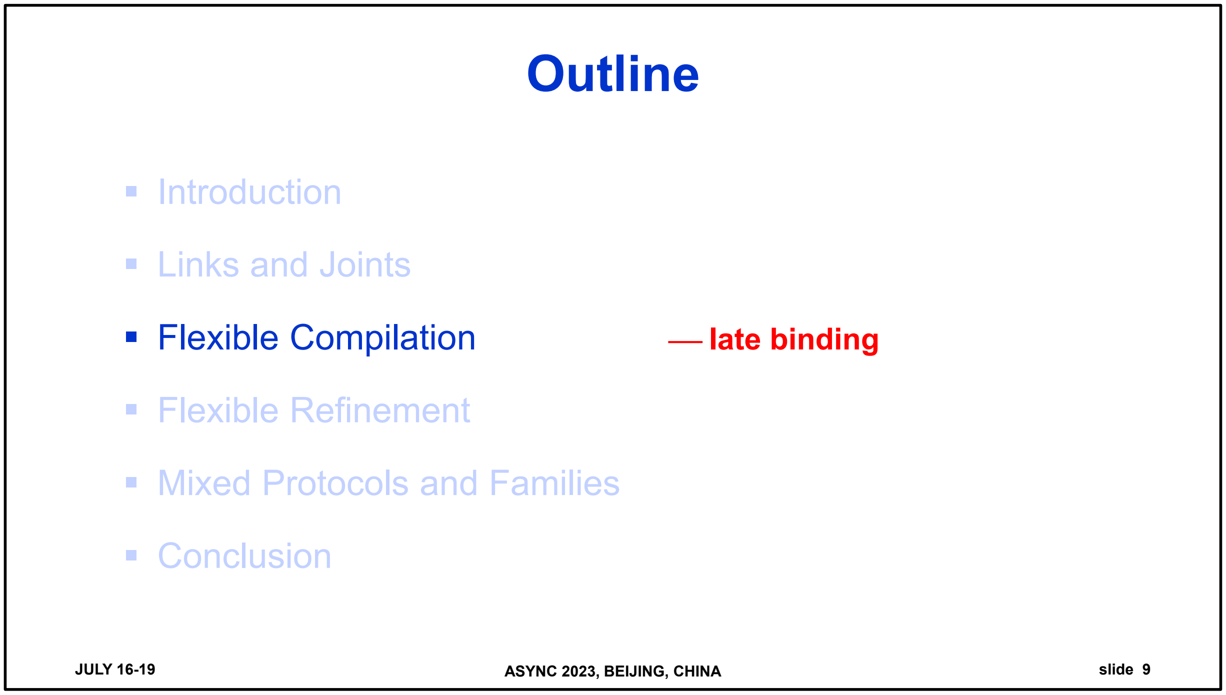
- In the same way, Joints TAKE TURNS updating the Link state

To model this, a Link has:

- two ports to attach Joints: port **A** and port **B**
- two data variables for data going from AtoB and BtoA, and
- a control variable called **turn** to store who has the current permission to change Link state: A or B.

Joints also have ports that connects to either Link port A or B

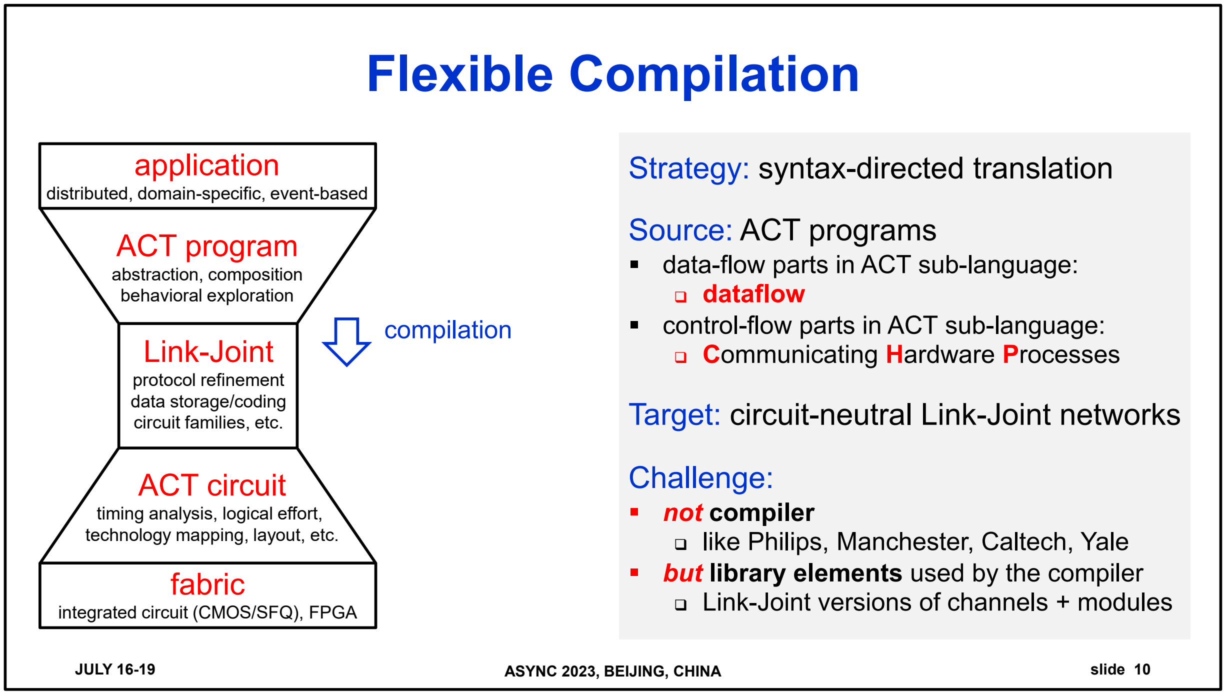
- The Joint port must have the turn to change the Link state.
- We specify Joints using guarded commands.



Outline

- Introduction
- Links and Joints
- Flexible Compilation — late binding
- Flexible Refinement
- Mixed Protocols and Families
- Conclusion

Now, Let’s look at compilation where late binding is made possible.



Flexible Compilation

application
distributed, domain-specific, event-based

ACT program
abstraction, composition
behavioral exploration

Link-Joint
protocol refinement
data storage/coding
circuit families, etc.

ACT circuit
timing analysis, logical effort,
technology mapping, layout, etc.

fabric
integrated circuit (CMOS/SFQ), FPGA

compilation

Strategy: syntax-directed translation

Source: ACT programs
- data-flow parts in ACT sub-language:
  - **dataflow**
- control-flow parts in ACT sub-language:
  - **C**ommunicating **H**ardware **P**rocesses

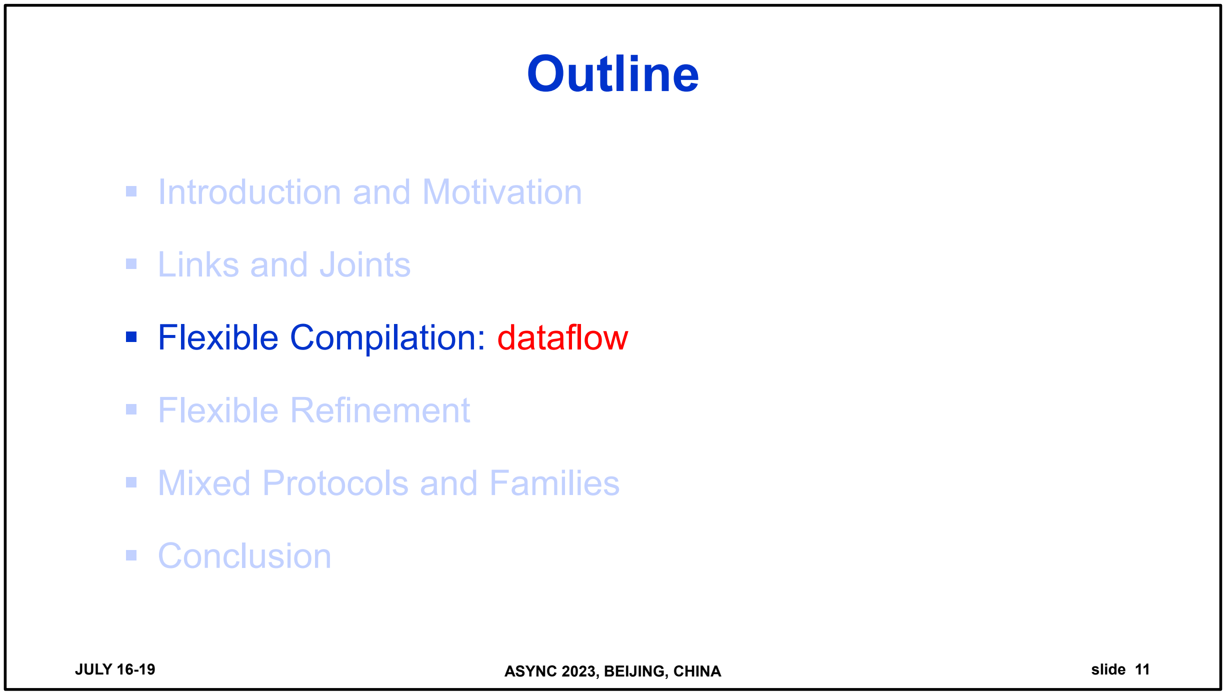
Target: circuit-neutral Link-Joint networks

Challenge:
- ***not*** **compiler**
  - like Philips, Manchester, Caltech, Yale
- ***but*** **library elements** used by the compiler
  - Link-Joint versions of channels + modules

The compiler uses syntax-directed translation.
It translates ACT programs, into circuit-neutral Link-Joint networks.

For us, the compilation challenge was **not the compiler itself**
- which is similar to what Philips, Manchester, Caltech, and Yale use.

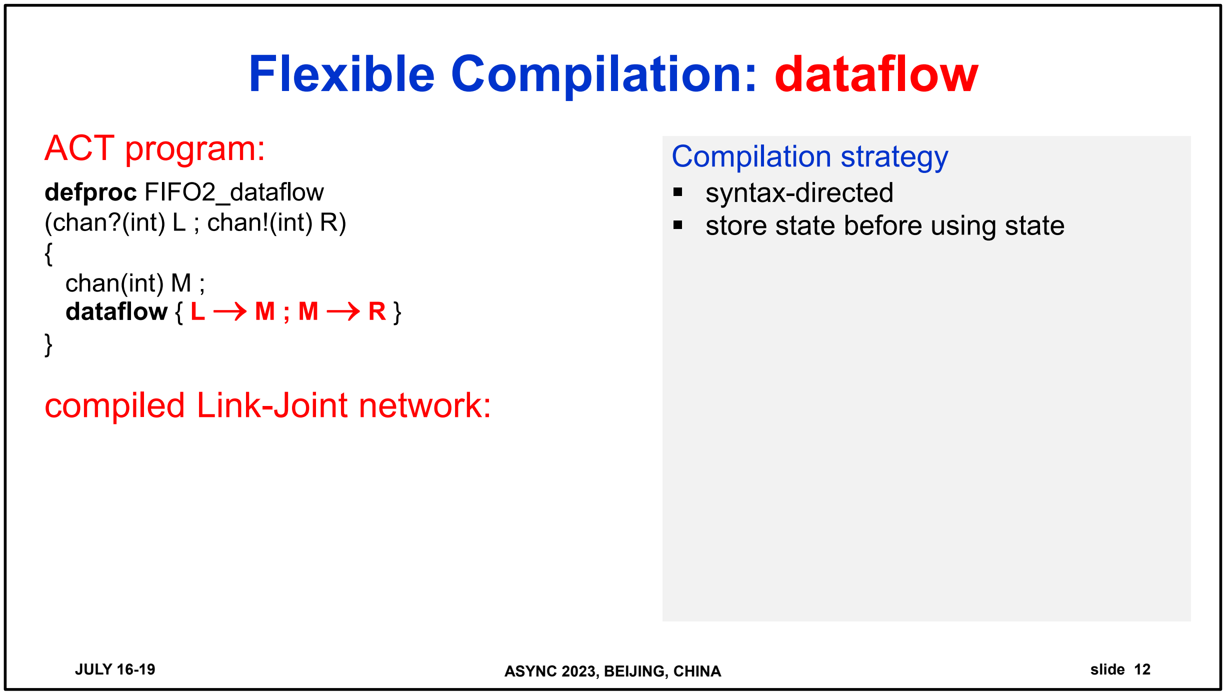
Rather, The challenge was to develop **the Link-Joint library elements**
- that the compiler generates
- instead of the traditional handshake channels and modules.



Outline

- Introduction and Motivation
- Links and Joints
- Flexible Compilation: dataflow
- Flexible Refinement
- Mixed Protocols and Families
- Conclusion

I will first show how we compile a dataflow program.



Flexible Compilation: dataflow

ACT program:

```
defproc FIFO2_dataflow
(chan?(int) L ; chan!(int) R)
{
   chan(int) M ;
   dataflow { L → M ; M → R }
}
```

compiled Link-Joint network:

Compilation strategy

- syntax-directed
- store state before using state

JULY 16-19 ASYNC 2023, BEIJING, CHINA slide 12

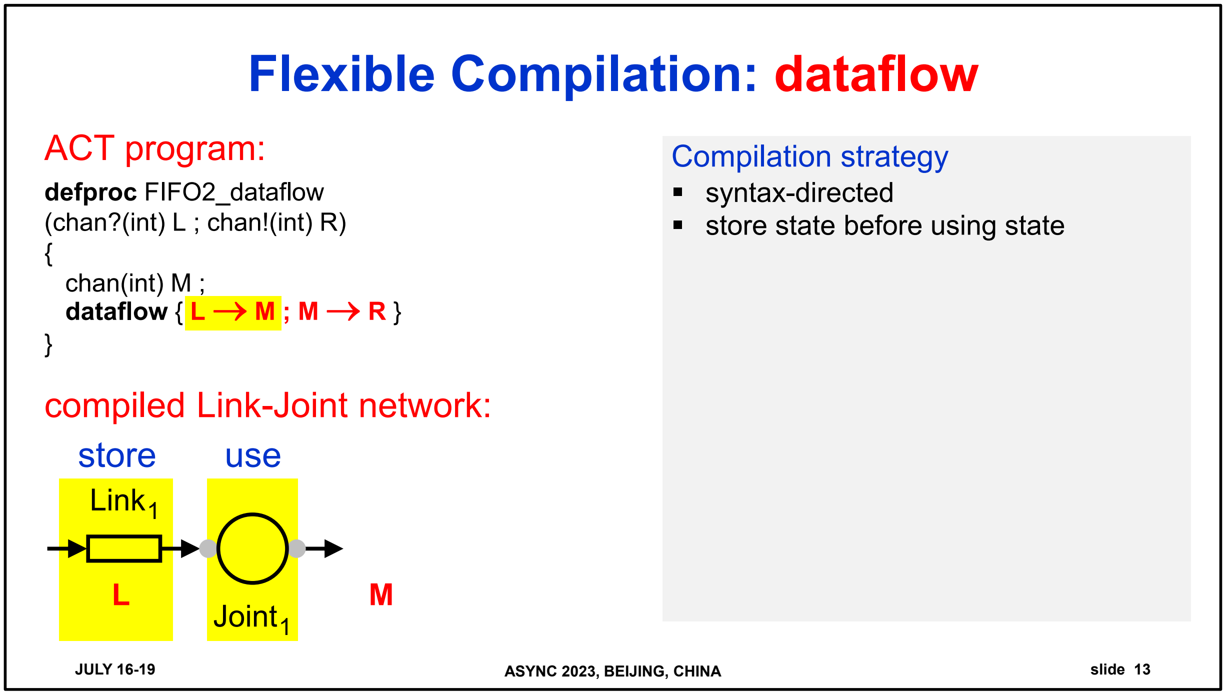
The ACT program on the left specifies a 2-stage FIFO with :

- input channel L
- and output channel R.

The core program uses a dataflow description.

Our compilation strategy for dataflow programs is to

- use syntax-directed translation
- and store state before we use the state.



Flexible Compilation: dataflow

ACT program:

```
defproc FIFO2_dataflow
(chan?(int) L ; chan!(int) R)
{
  chan(int) M ;
  dataflow { L → M ; M → R }
}
```

compiled Link-Joint network:

Compilation strategy

- syntax-directed
- store state before using state

The first dataflow command

- “L arrow M”

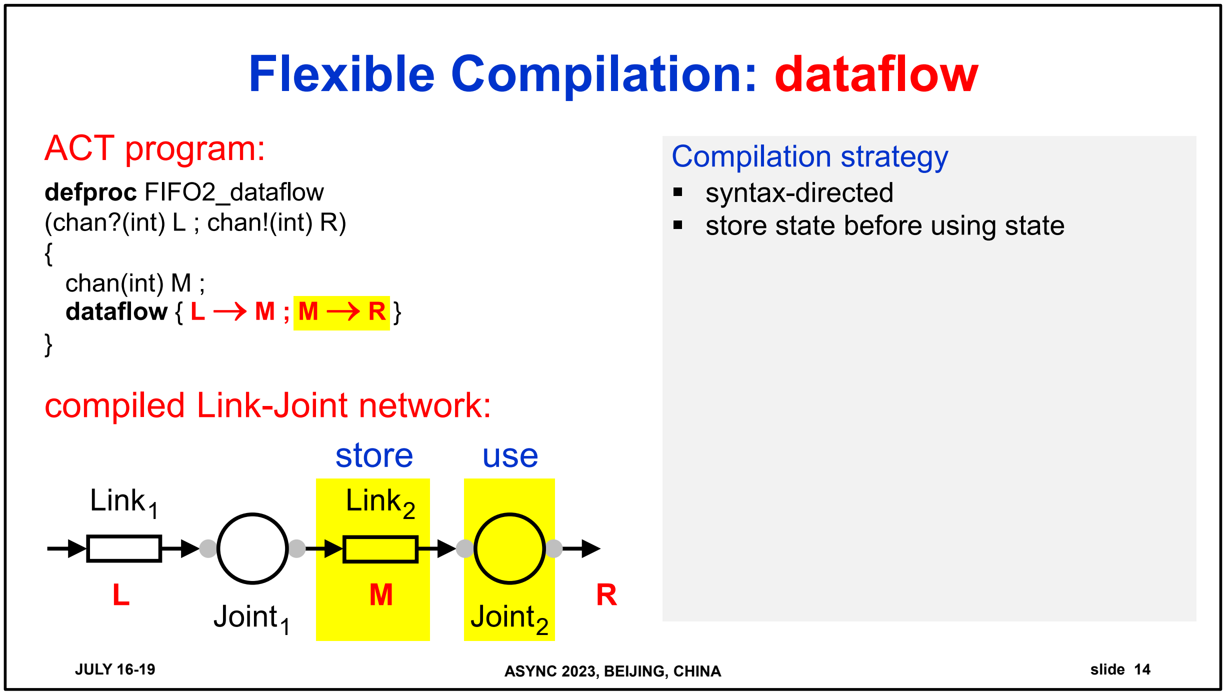
stands for

- “copy data from external channel L to local channel M”

To compile this command:

- we store the data on L, using Link1
- and then we copy the data to M using Joint1.

<PAUSE for 2 seconds>



Flexible Compilation: dataflow

ACT program:

```
defproc FIFO2_dataflow
(chan?(int) L ; chan!(int) R)
{
   chan(int) M ;
   dataflow { L → M ; M → R }
}
```

compiled Link-Joint network:

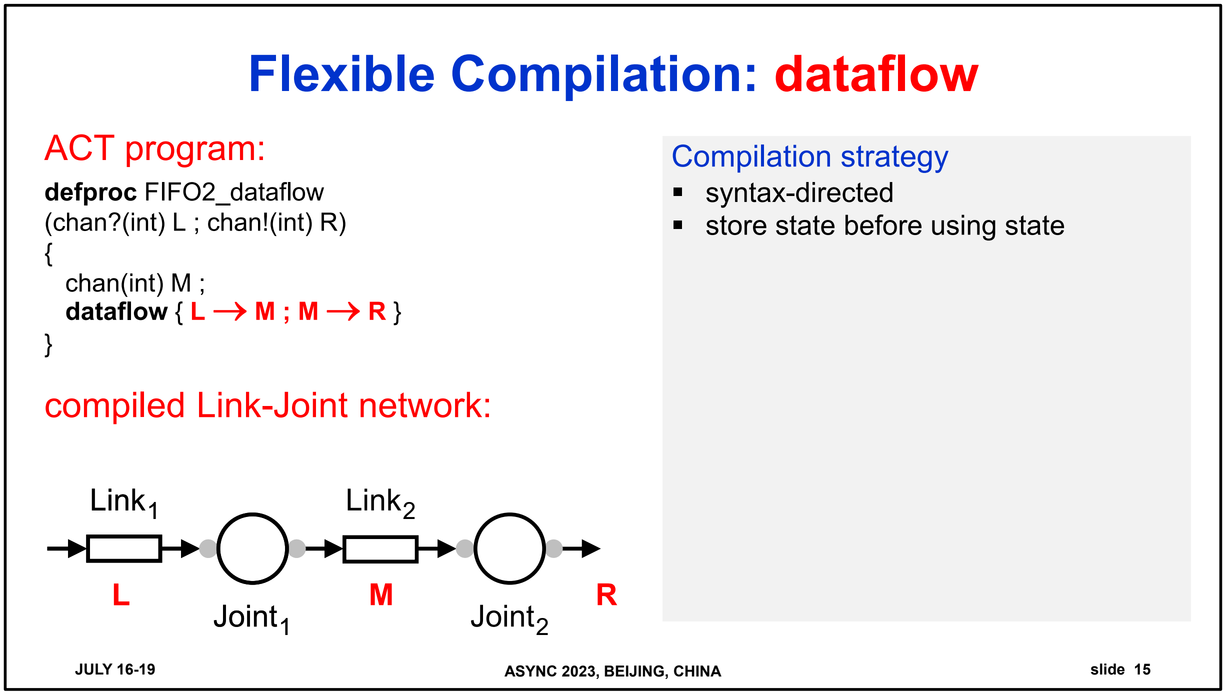
Compilation strategy

- syntax-directed
- store state before using state

Likewise for the second command “M arrow R”

- we store the data on M using Link2
- and then we copy the data to R using Joint2.

<PAUSE for 2 seconds>



Flexible Compilation: dataflow

ACT program:

```
defproc FIFO2_dataflow
(chan?(int) L ; chan!(int) R)
{
  chan(int) M ;
  dataflow { L → M ; M → R }
}
```

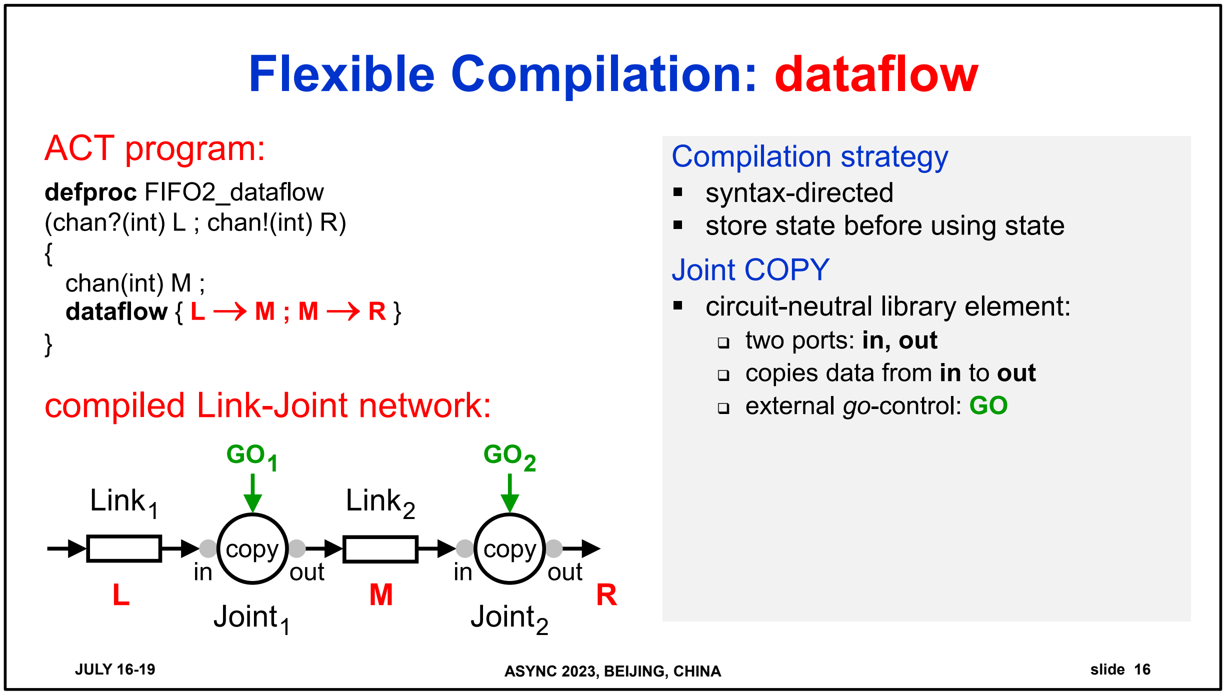
compiled Link-Joint network:

Compilation strategy

- syntax-directed
- store state before using state

Here is the resulting Link-Joint network.

Now let's talk about the library elements for Joint1 and Joint2.



Flexible Compilation: dataflow

ACT program:

```
defproc FIFO2_dataflow
(chan?(int) L ; chan!(int) R)
{
  chan(int) M ;
  dataflow { L → M ; M → R }
}
```

compiled Link-Joint network:

Compilation strategy

- syntax-directed
- store state before using state

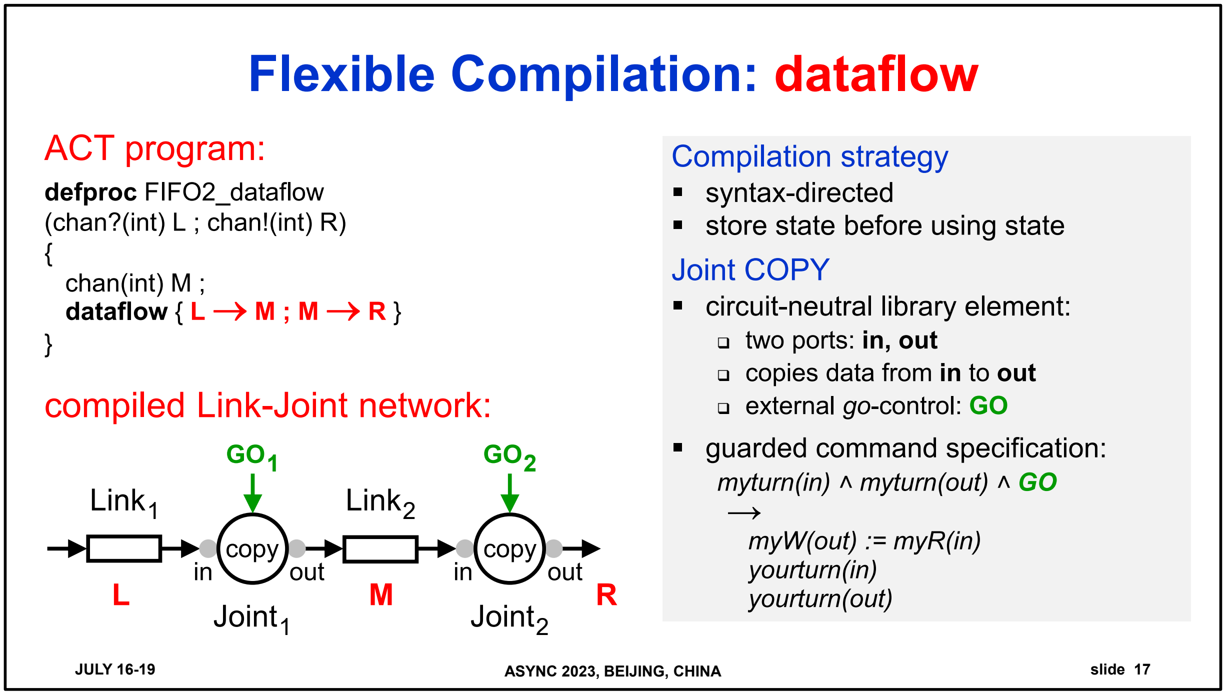
Joint COPY

- circuit-neutral library element:
  - two ports: **in, out**
  - copies data from **in** to **out**
  - external *go*-control: **GO**

Both Joints copy data from their input Link to their output Link.

So, we use Joint COPY, which is

- a circuit-neutral library element
- with two ports, in and out
- and an external go-control signal for initialization and test.



Flexible Compilation: dataflow

ACT program:

```
defproc FIFO2_dataflow
(chan?(int) L ; chan!(int) R)
{
  chan(int) M ;
  dataflow { L → M ; M → R }
}
```

compiled Link-Joint network:

Compilation strategy

- syntax-directed
- store state before using state

Joint COPY

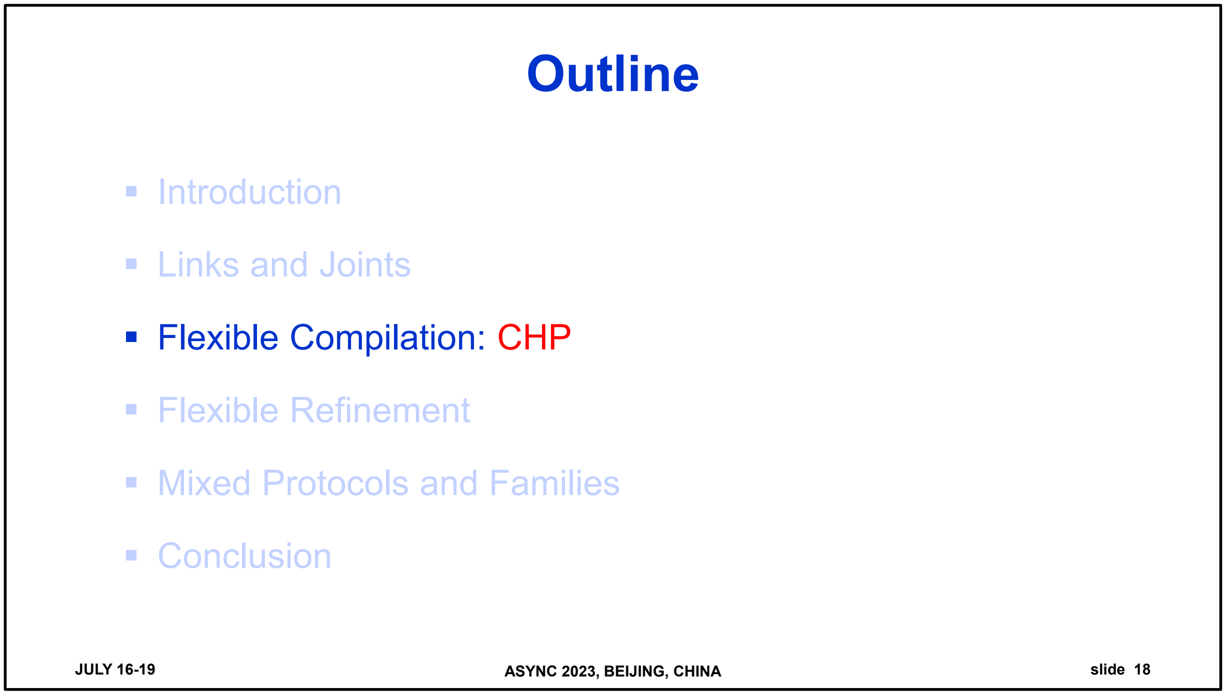
- circuit-neutral library element:
  - two ports: **in, out**
  - copies data from **in** to **out**
  - external *go*-control: **GO**
- guarded command specification:

$$\begin{aligned}&\mathit{myturn(in)} \wedge \mathit{myturn(out)} \wedge \mathbf{GO}\\ &\longrightarrow\\ &\quad \mathit{myW(out)} := \mathit{myR(in)}\\ &\quad \mathit{yourturn(in)}\\ &\quad \mathit{yourturn(out)}\end{aligned}$$

Here is the guarded command specification of Joint COPY.

<POINT-TO guarded command>
- When both ports in and out have the turn
- and the Joint has go-permission
- then
- write out the data you read in
- and give both turns back.

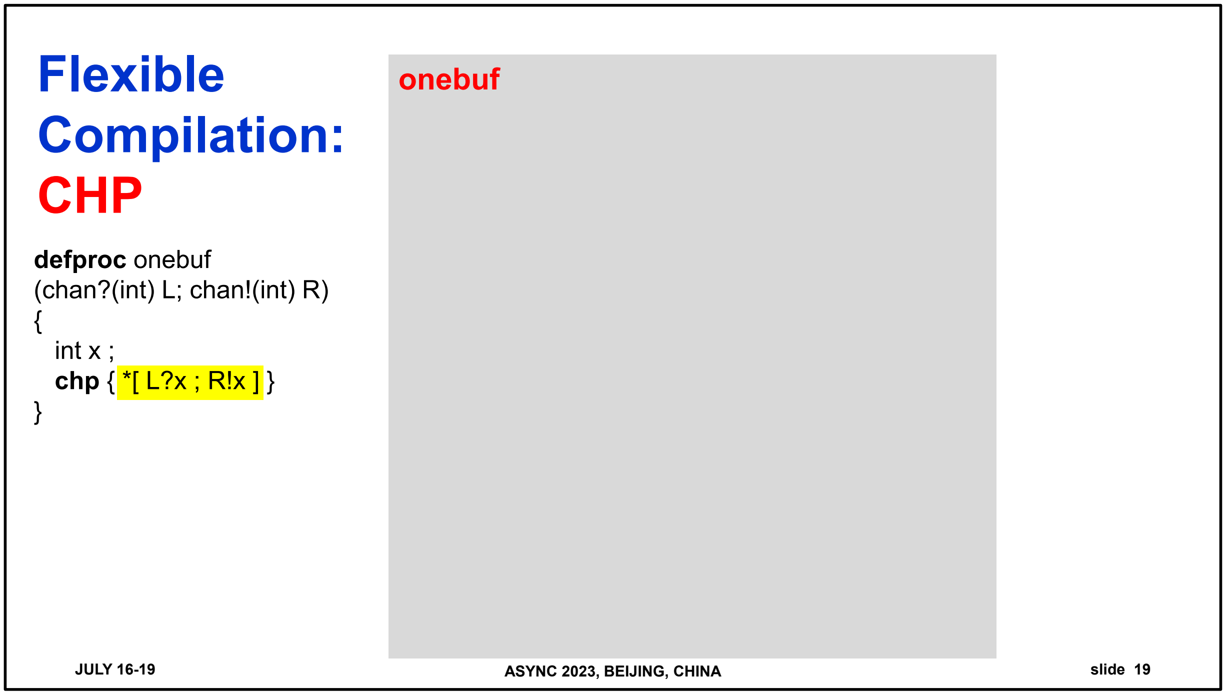
<PAUSE 2 seconds>



Outline

- Introduction
- Links and Joints
- Flexible Compilation: CHP
- Flexible Refinement
- Mixed Protocols and Families
- Conclusion

That was dataflow, now let’s look at control flow.



Flexible Compilation: CHP

```
defproc onebuf
(chan?(int) L; chan!(int) R)
{
  int x ;
  chp { *[ L?x ; R!x ] }
}
```

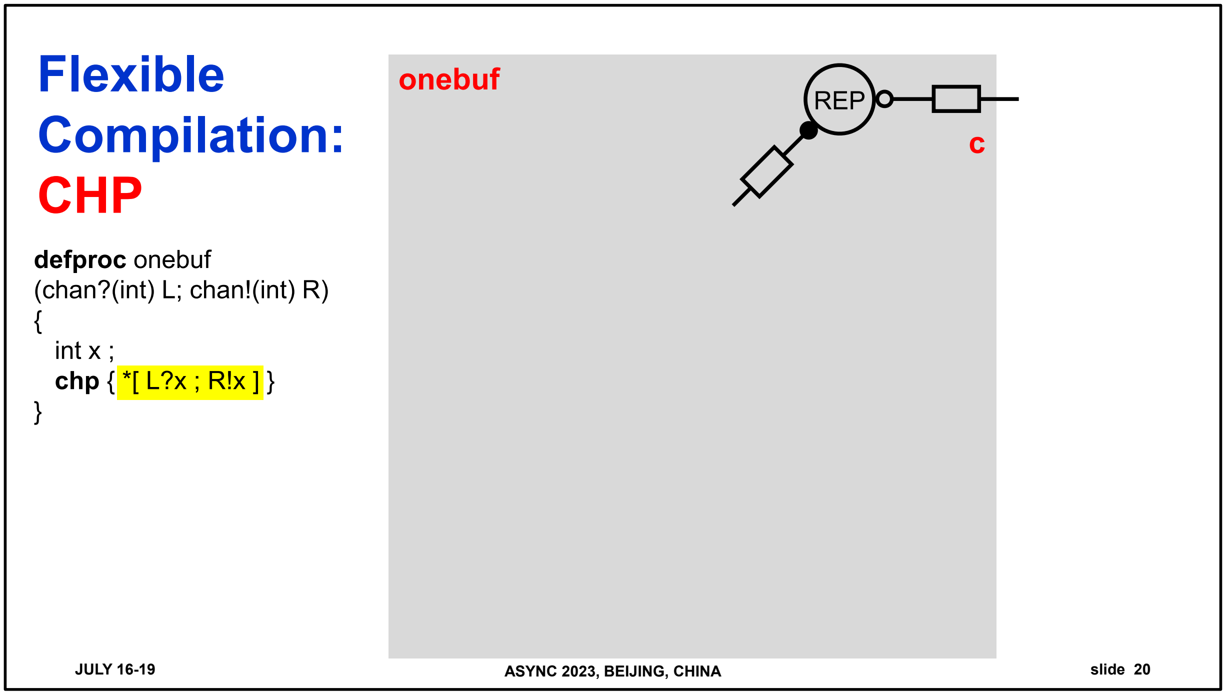
onebuf

The ACT program **onebuf** on the left specifies a 1-stage FIFO with :

- input channel L
- and output channel R.

The core program uses a CHP control-flow description.

The syntax-directed compilation goes as follows.



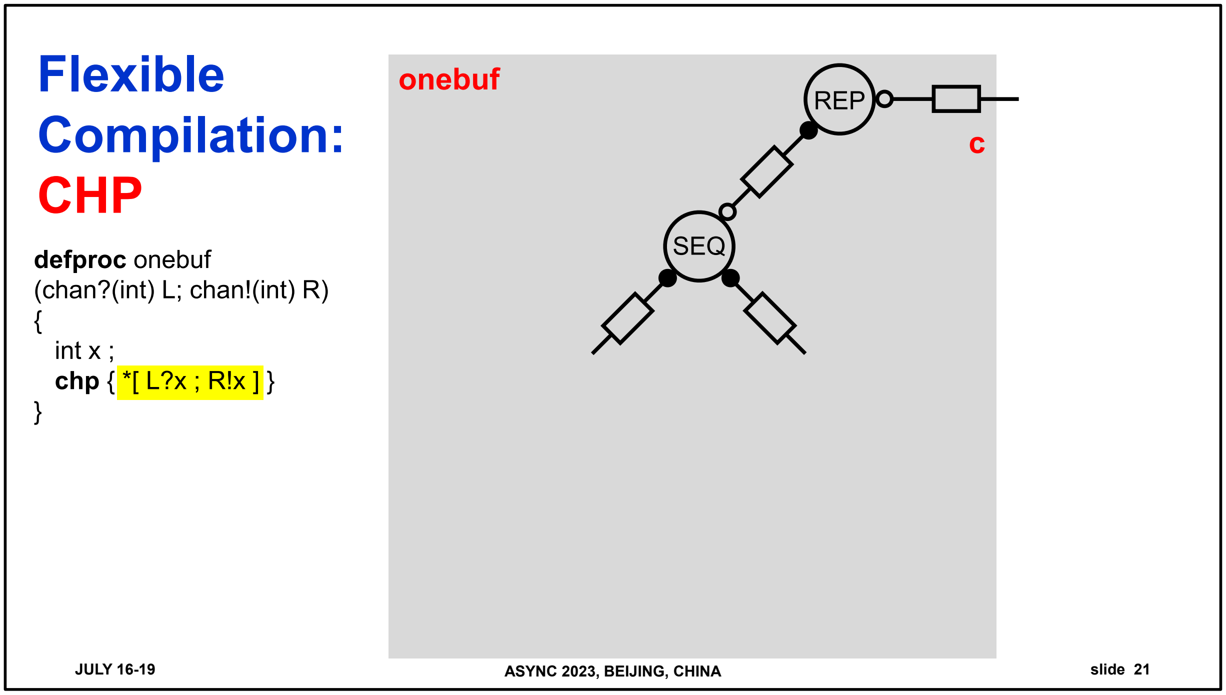
Flexible Compilation: CHP

```
defproc onebuf
(chan?(int) L; chan!(int) R)
{
  int x ;
  chp { *[ L?x ; R!x ] }
}
```

JULY 16-19 ASYNC 2023, BEIJING, CHINA slide 20

We translate

- the “star” in the program – which represents repetition
- to a REPEAT Joint with startup Link c.

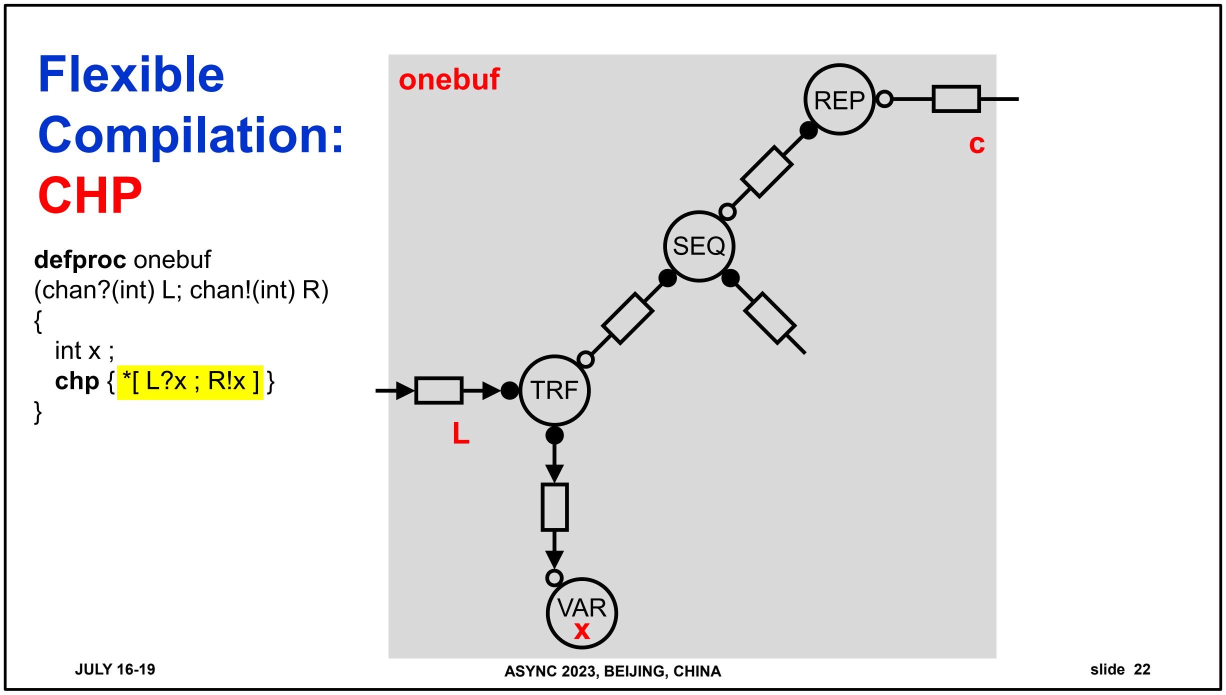


Flexible Compilation: CHP

```
defproc onebuf
(chan?(int) L; chan!(int) R)
{
  int x ;
  chp { *[ L?x ; R!x ] }
}
```

We translate

- the “semicolon” – which represents sequential composition
- to a SEQUENCE Joint with Links and connect it to the REPEAT Joint.



Flexible Compilation: CHP

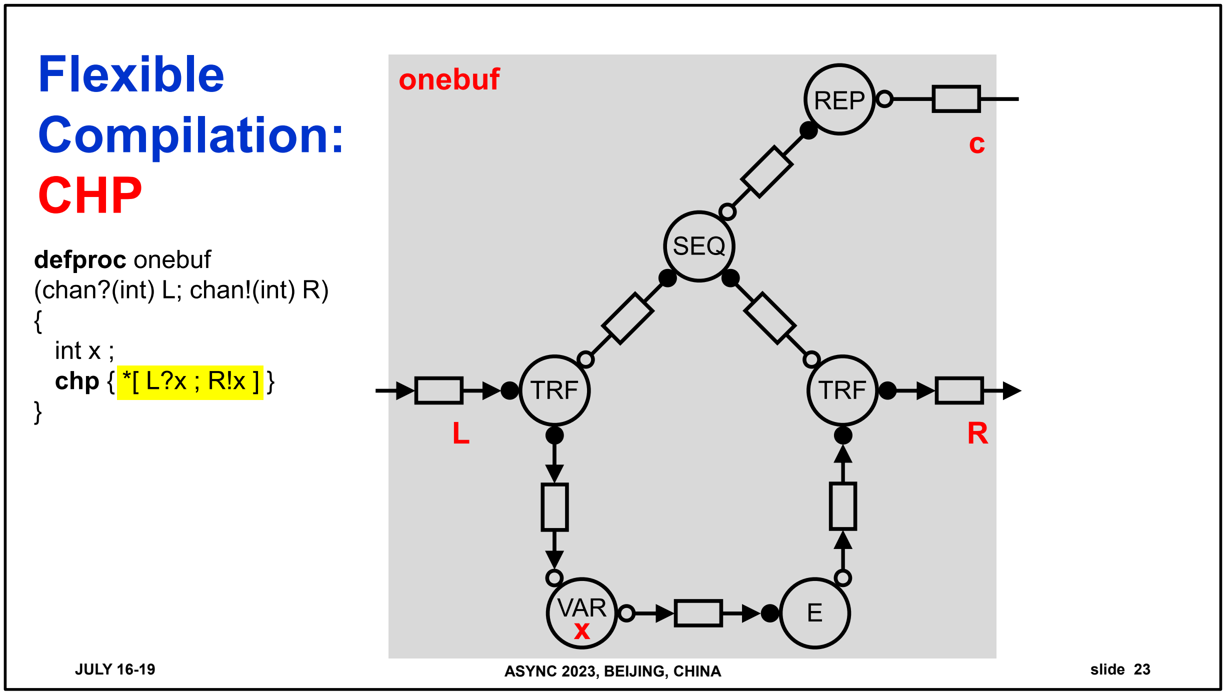
```
defproc onebuf
(chan?(int) L; chan!(int) R)
{
  int x ;
  chp { *[ L?x ; R!x ] }
}
```

JULY 16-19 ASYNC 2023, BEIJING, CHINA slide 22

For the first branch of the SEQUENCE Joint,
We translate the first sequential statement “L question mark x.”

This is an input communication
for which we create a communication network

- that transfers input data from channel L
- to variable x



Flexible Compilation: CHP

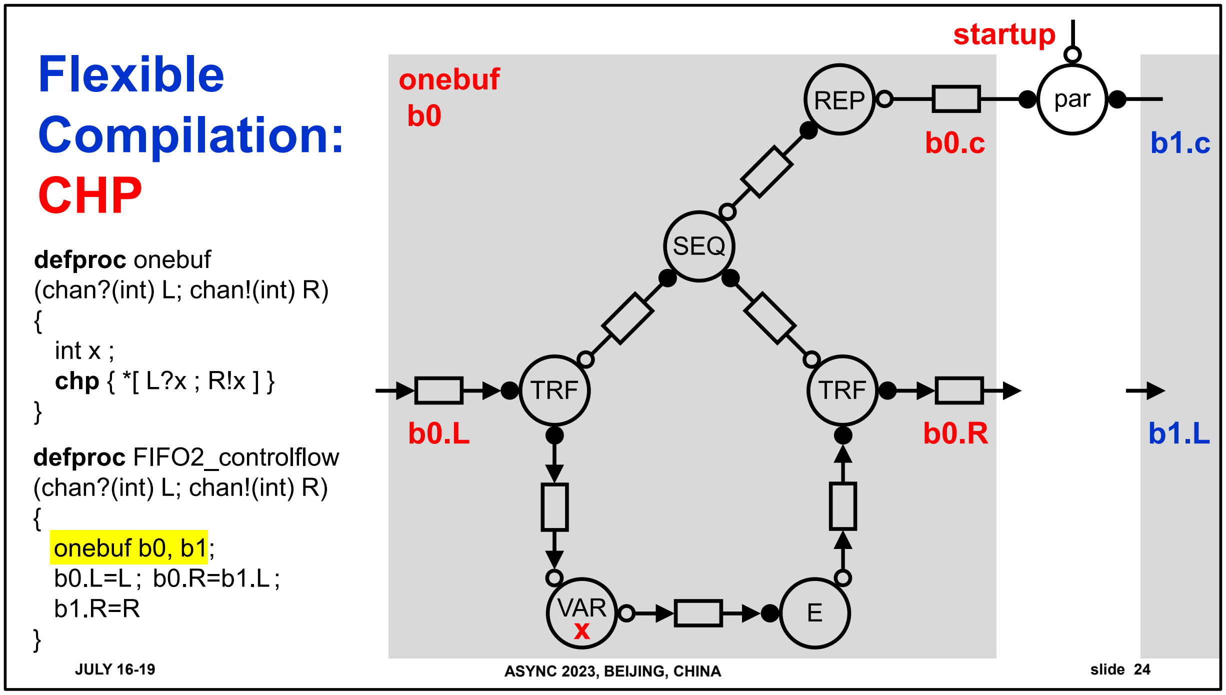
```
defproc onebuf
(chan?(int) L; chan!(int) R)
{
  int x ;
  chp { *[ L?x ; R!x ] }
}
```

JULY 16-19 ASYNC 2023, BEIJING, CHINA slide 23

Likewise for the second branch of the SEQUENCE Joint,
we use a similar translation for the second sequential statement
“R exclamation mark x.”

This is an output communication
for which we create a communication network

- that transfers data from variable x
- to channel R.



Flexible Compilation: CHP

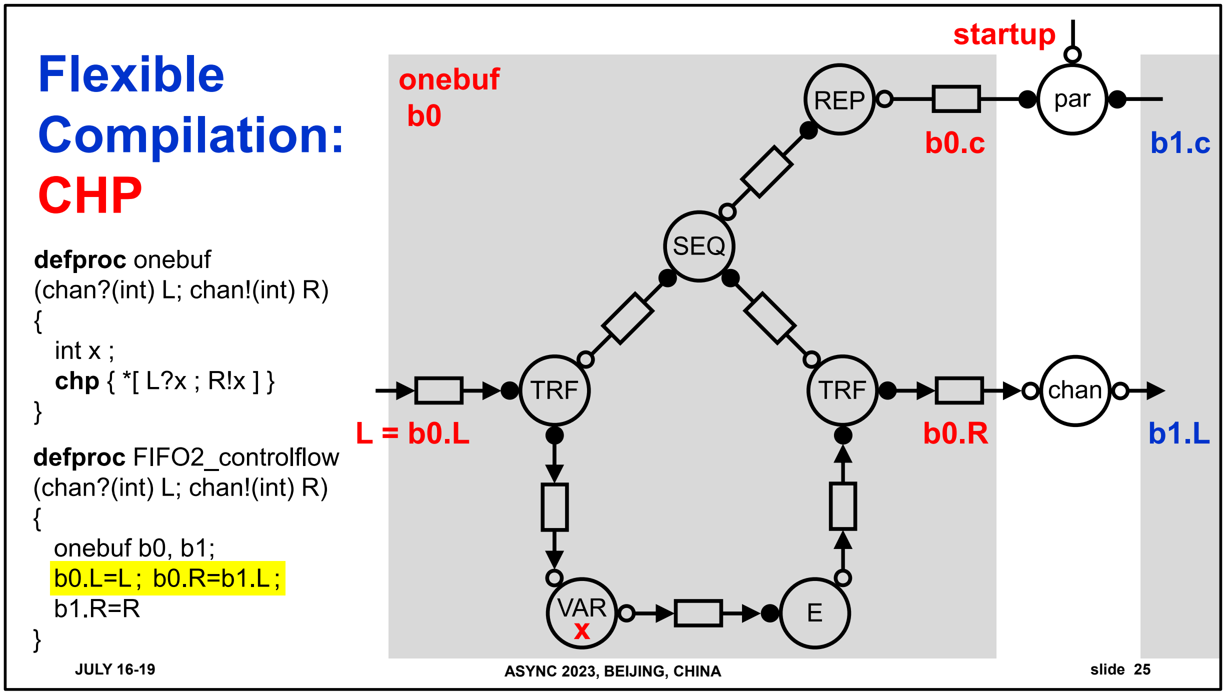
```
defproc onebuf
(chan?(int) L; chan!(int) R)
{
  int x ;
  chp { *[ L?x ; R!x ] }
}
defproc FIFO2_controlflow
(chan?(int) L; chan!(int) R)
{
  onebuf b0, b1;
  b0.L=L ;  b0.R=b1.L ;
  b1.R=R
}
```

JULY 16-19 ASYNC 2023, BEIJING, CHINA slide 24

With ACT, we can create a 2-stage control-flow FIFO, by using two instances of process **onebuf** , b0 and b1. This is done in the lower ACT program on the left.

The compiler connects the startup Links of b0 and b1

- with a PARALLEL Joint <POINT-TO par>
- to ensure that the compiled instances are executed in parallel.



Flexible Compilation: CHP

```
defproc onebuf
(chan?(int) L; chan!(int) R)
{
  int x ;
  chp { *[ L?x ; R!x ] }
}
defproc FIFO2_controlflow
(chan?(int) L; chan!(int) R)
{
  onebuf b0, b1;
  b0.L=L ; b0.R=b1.L ;
  b1.R=R
}
```

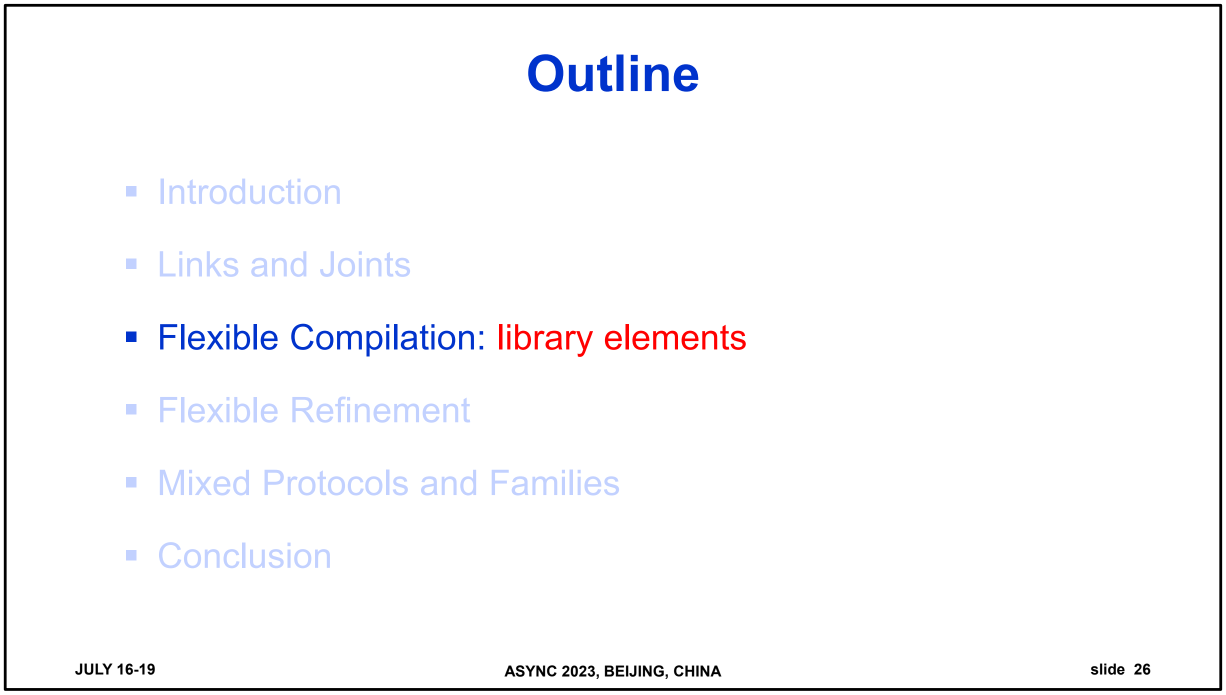
JULY 16-19 ASYNC 2023, BEIJING, CHINA slide 25

The compiler follows the aliasing commands in the ACT program.

The compiler also connects parallel Links for the same channel

- with a CHANNEL Joint
- to ensure that the compiled channels
- can be probed and synchronized.

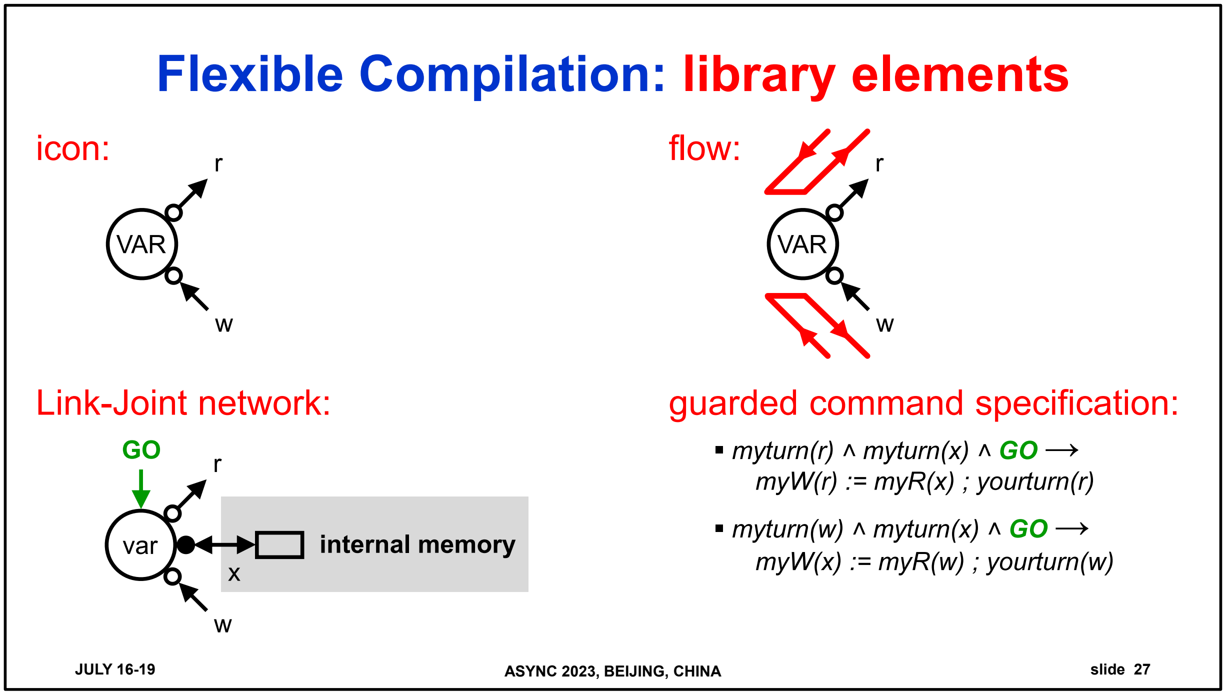
Notice that this slide has no circuits in it.
We have not made any decisions about protocols, data encodings or circuit families.
We never make decisions before it is time to make them. This is called late binding.
This is what our approach is about.



Outline

- Introduction
- Links and Joints
- Flexible Compilation: library elements
- Flexible Refinement
- Mixed Protocols and Families
- Conclusion

As I said earlier, the library elements were our compilation challenge.
So, I will give two examples to show how we address this.



Flexible Compilation: library elements

icon:

flow:

Link-Joint network:

guarded command specification:

- *myturn(r) ∧ myturn(x) ∧* **GO** ⟶
  myW(r) := myR(x) ; yourturn(r)
- *myturn(w) ∧ myturn(x) ∧* **GO** ⟶
  myW(x) := myR(w) ; yourturn(w)

<point to icon>
The first example is the Joint for a CHP variable with read and write ports.

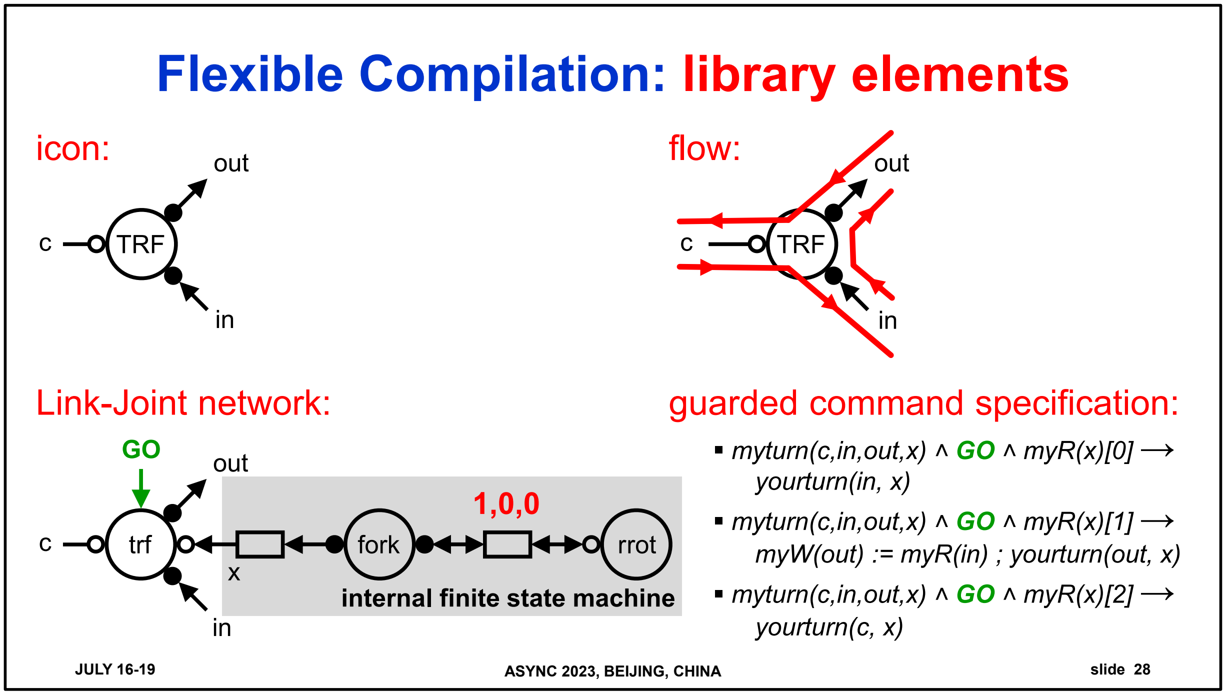
In the Link-Joint model, only Links store state.
Therefore, the Link-Joint network for this variable

- has an internal Link with port x to store the value of the variable.

The guarded command specifies that

- read and write are mutual exclusive
- and that we read from and write to Link port x.

<WAIT 2 seconds>



Flexible Compilation: library elements

guarded command specification:

- *myturn(c,in,out,x)* ∧ **GO** ∧ *myR(x)[0]* ⟶ *yourturn(in, x)*
- *myturn(c,in,out,x)* ∧ **GO** ∧ *myR(x)[1]* ⟶ *myW(out) := myR(in) ; yourturn(out, x)*
- *myturn(c,in,out,x)* ∧ **GO** ∧ *myR(x)[2]* ⟶ *yourturn(c, x)*

JULY 16-19 ASYNC 2023, BEIJING, CHINA slide 28

The second example is Joint TRANSFER. We use it as a local router of data in the network.
it has sequential behavior.

- We sequence its operations with a finite state machine

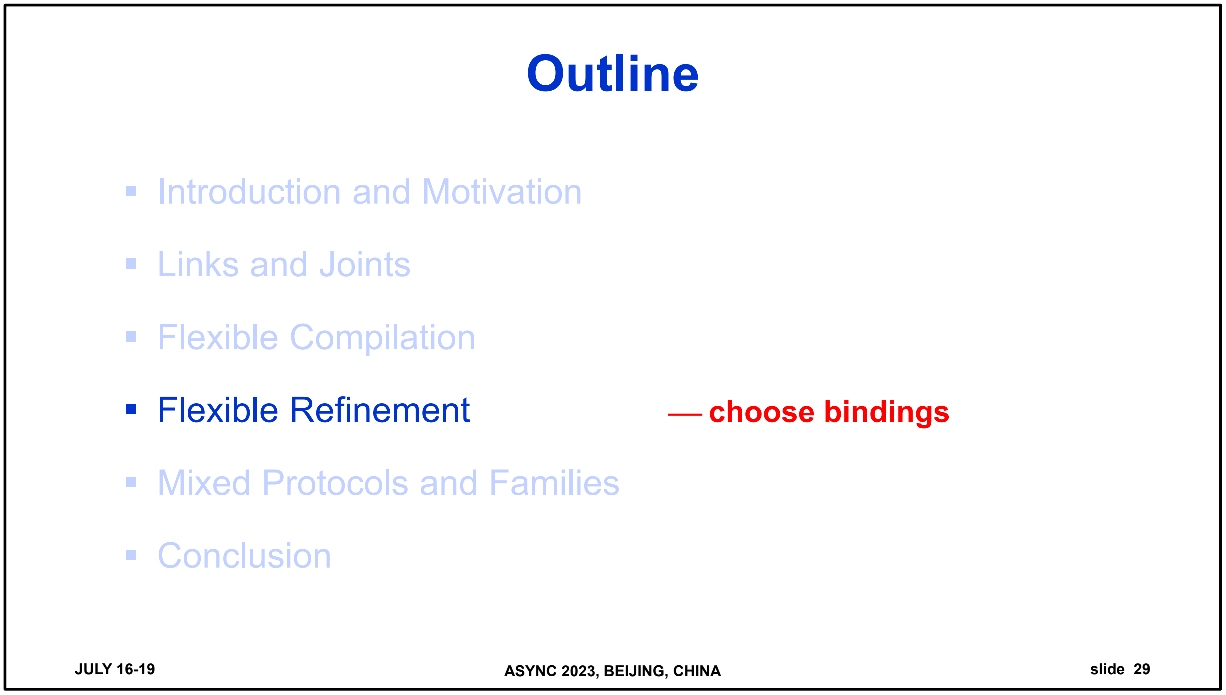
This finite state machine generates a 1-hot code that determines
which of the 3 commands of TRANSFER to execute

The flow diagram on the right shows that the operation of<FOLLOW flow diagram>

- TRANSFER starts at port c, proceeds to in, then to out, and back to c.

<PAUSE 2 seconds>

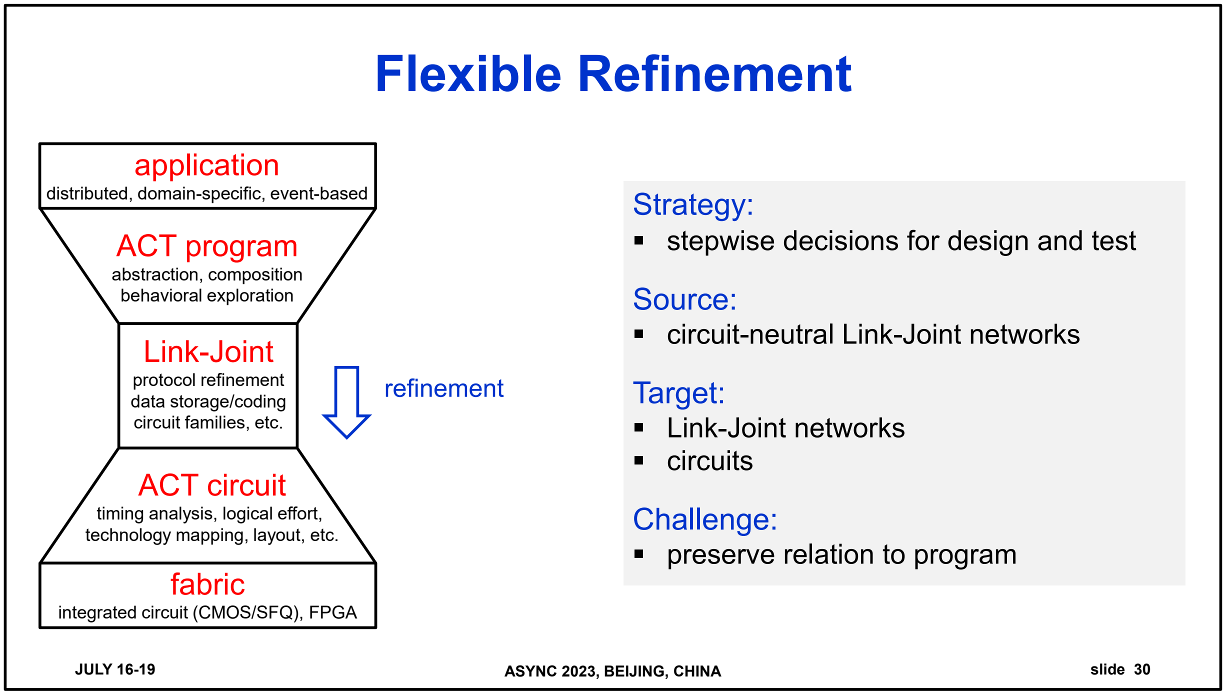
Our paper has more library elements, with their flow diagrams and guarded commands.



Outline

- Introduction and Motivation
- Links and Joints
- Flexible Compilation
- Flexible Refinement — choose bindings
- Mixed Protocols and Families
- Conclusion

That was compilation, now let’s look at refinement where we choose bindings.



Flexible Refinement

application
distributed, domain-specific, event-based

ACT program
abstraction, composition
behavioral exploration

Link-Joint
protocol refinement
data storage/coding
circuit families, etc.

ACT circuit
timing analysis, logical effort,
technology mapping, layout, etc.

fabric
integrated circuit (CMOS/SFQ), FPGA

refinement

Strategy:
- stepwise decisions for design and test

Source:
- circuit-neutral Link-Joint networks

Target:
- Link-Joint networks
- circuits

Challenge:
- preserve relation to program

JULY 16-19 ASYNC 2023, BEIJING, CHINA slide 30

Refinement is a process of stepwise decisions about
- which protocols, data encodings, and circuit families to use
- and when and where to store data

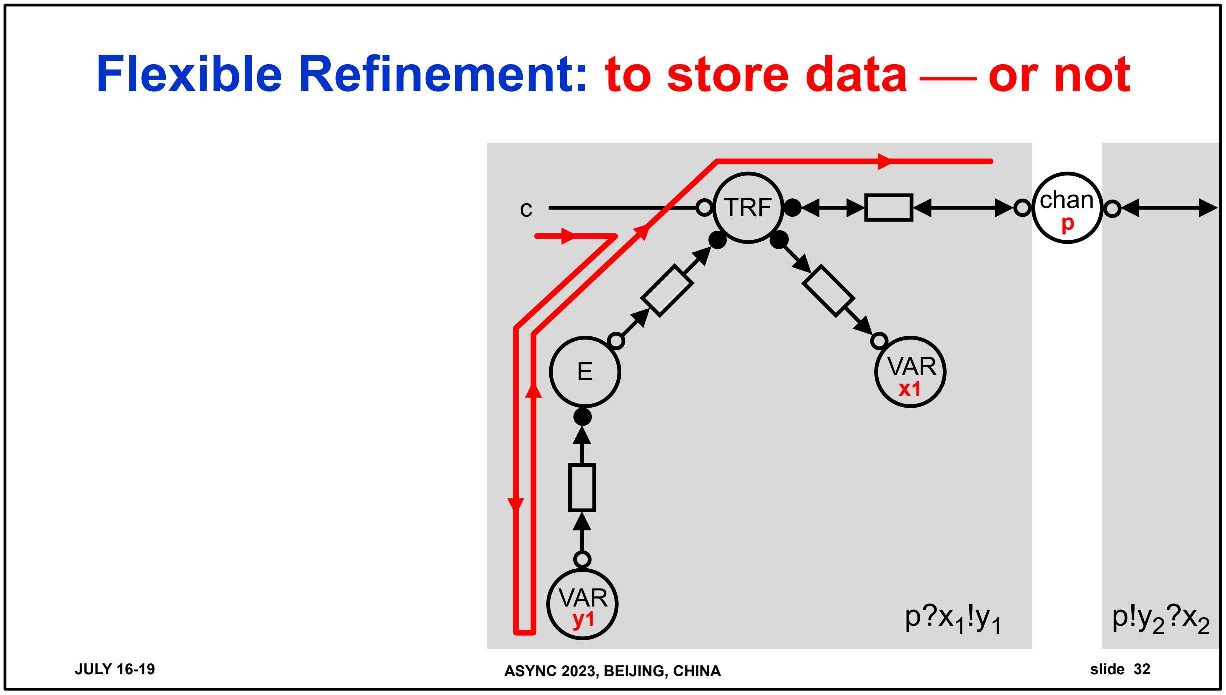
for design and test purposes.

Starting with circuit-neutral Link-Joint networks, the refinement process ends with circuits.

The challenge here is to preserve the relation to the original ACT program.

Flexible Refinement: to store data — or not

One important refinement decision is: do we store data – or not?
This is especially important when there are many data bits, because storage is expensive in delay, area, and power.



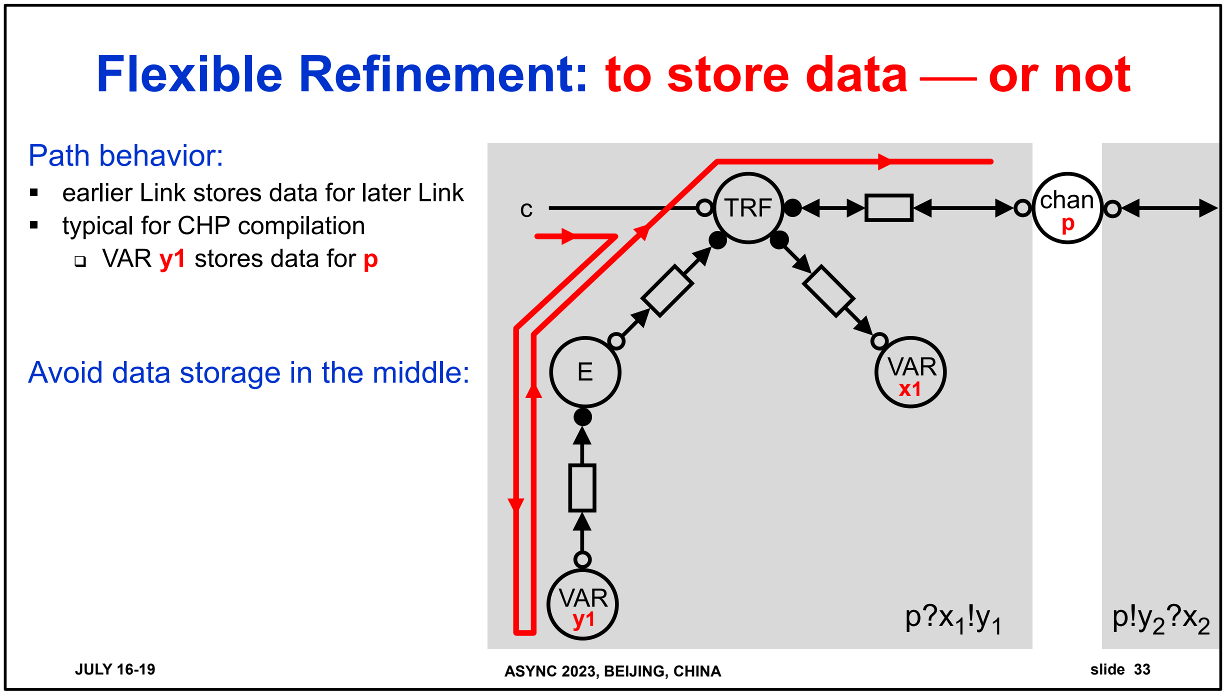
Flexible Refinement: to store data — or not

JULY 16-19 ASYNC 2023, BEIJING, CHINA slide 32

On the right, is an example of a bidirectional CHP communication.
The left side of this communication: <POINT TO p?x1!y1>

- Requests for a value over channel p for x1
- and sends the value of y1 over p – at the same time.

The right side of the communication is symmetric.
We can say that this network exchanges data between both sides.



Flexible Refinement: to store data — or not

Path behavior:

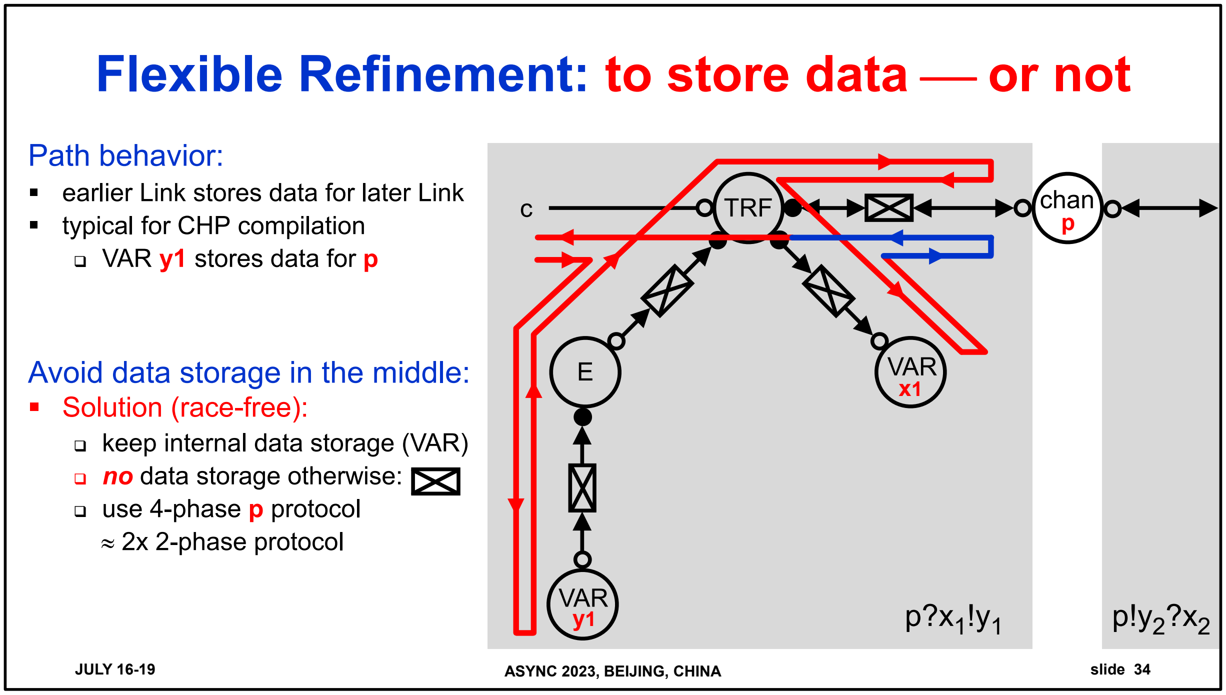
- earlier Link stores data for later Link
- typical for CHP compilation
  - VAR **y1** stores data for **p**

Avoid data storage in the middle:

CHP translations often leads to a path behavior where an earlier Link stores data for a later Link in the path.
In this example: <FOLLOW PATH FROM y1 to p>

- the internal Link in variable y1 stores data
- To be sent to the other side of channel p.

Because y1 stores data for p, the Links between them don’t have to.
The paper gives 3 solutions to remove data storage. I’ll show one of these 3.



Flexible Refinement: to store data — or not

Path behavior:

- earlier Link stores data for later Link
- typical for CHP compilation
  - VAR **y1** stores data for **p**

Avoid data storage in the middle:

- Solution (race-free):
  - keep internal data storage (VAR)
  - ***no*** data storage otherwise: ☒
  - use 4-phase **p** protocol
    ≈ 2x 2-phase protocol

JULY 16-19 ASYNC 2023, BEIJING, CHINA slide 34

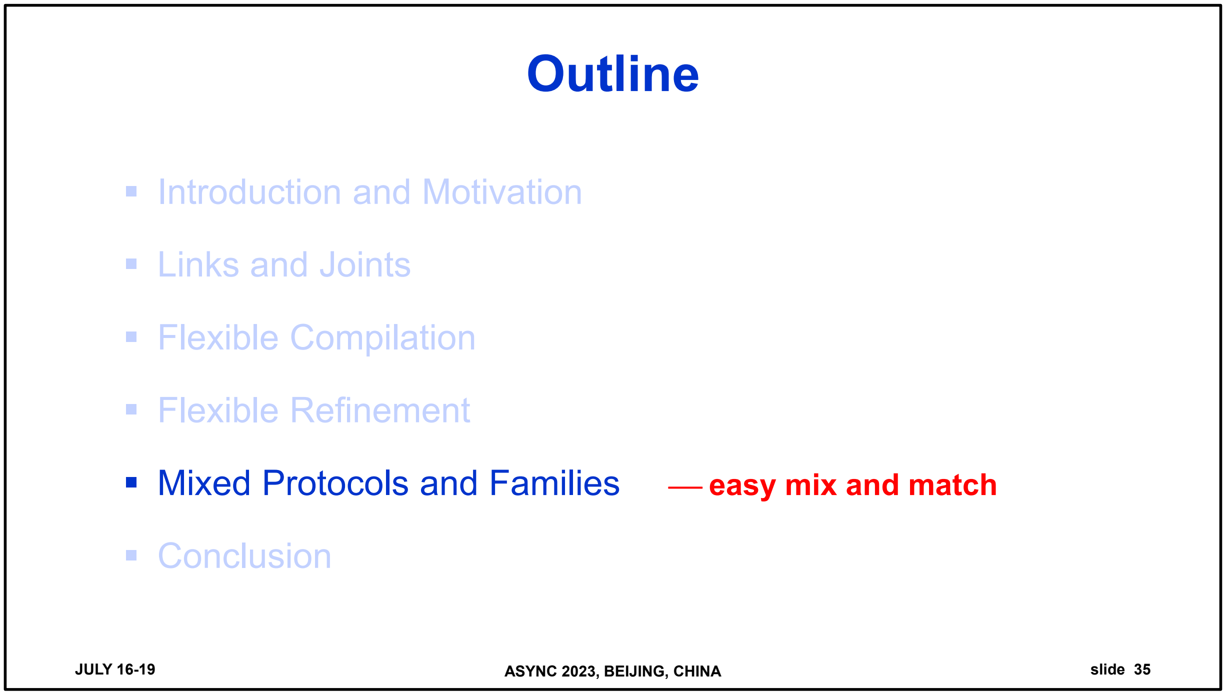
Here is a race-free solution for this example.

- Only the variables store data
- All other Links avoid storing data – the Links without data storage are shown with a crossed box.
- In addition, channel p uses a 4-phase communication protocol.

During phase 1 and 2 <FOLLOW top-right path> the protocol synchronizes the exchange of y1 and y2 values
While in phase 3 and 4 <FOLLOW bottom-right path>
the protocol synchronizes the completion of the data storage in x1 and x2 for both sides of channel p
This means that both sides store the values they receive before they end their communication.

This 4-phase protocol can be implemented as two 2-phase protocols.
As I mentioned earlier, This is one of 3 data storage elimination solutions presented in the paper.

You can also read about other refinements in our paper.



Outline

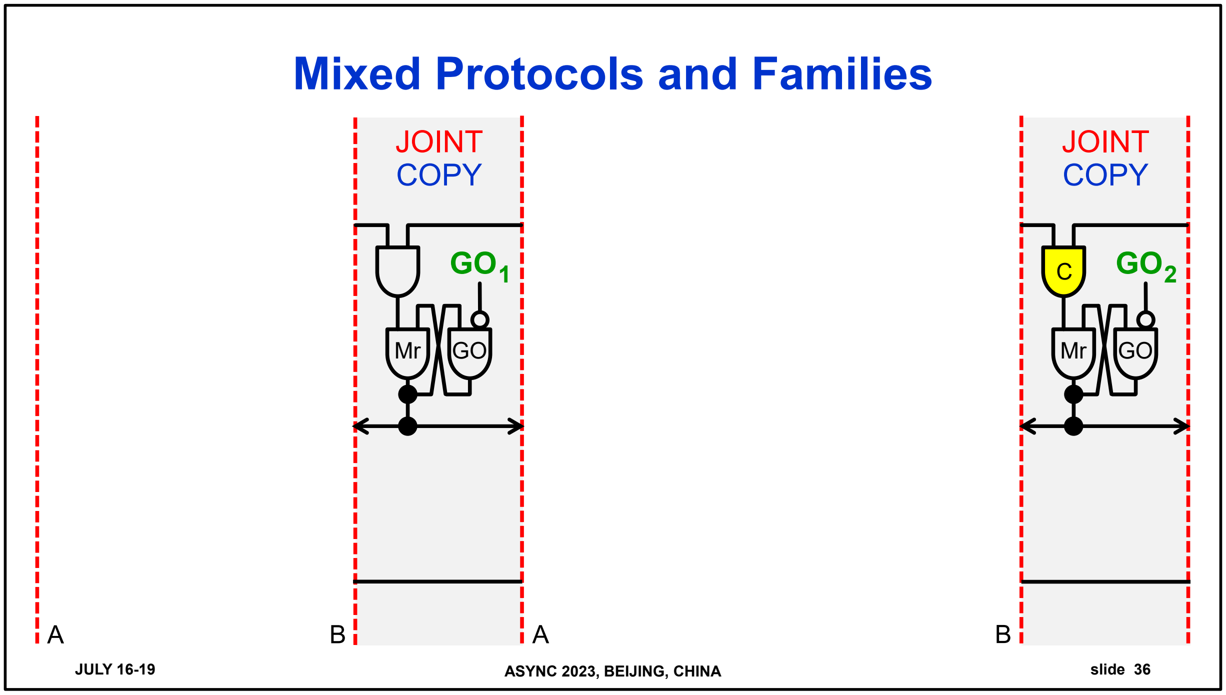
- Introduction and Motivation
- Links and Joints
- Flexible Compilation
- Flexible Refinement
- **Mixed Protocols and Families** — **easy mix and match**
- Conclusion

The compilation and refinements that I showed you

- maintain key features of Links and Joints.

In particular:

- they allow the use of mixed protocols and circuit families.



Mixed Protocols and Families

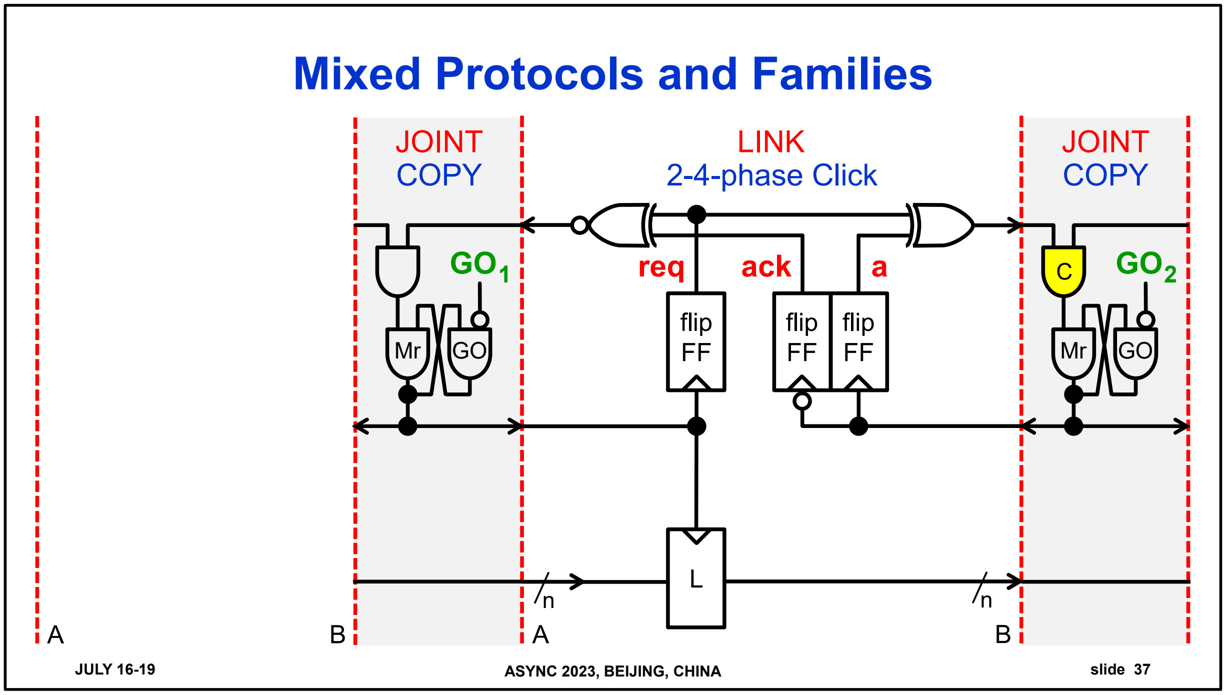
JULY 16-19

ASYNC 2023, BEIJING, CHINA

slide 36

Here is an example.
I am showing two different circuit implementations of Joint COPY.

The Joint with an AND gate on the left uses 2 phase protocols
while The Joint with a C element on the right uses 4 phase protocols



Mixed Protocols and Families

JULY 16-19

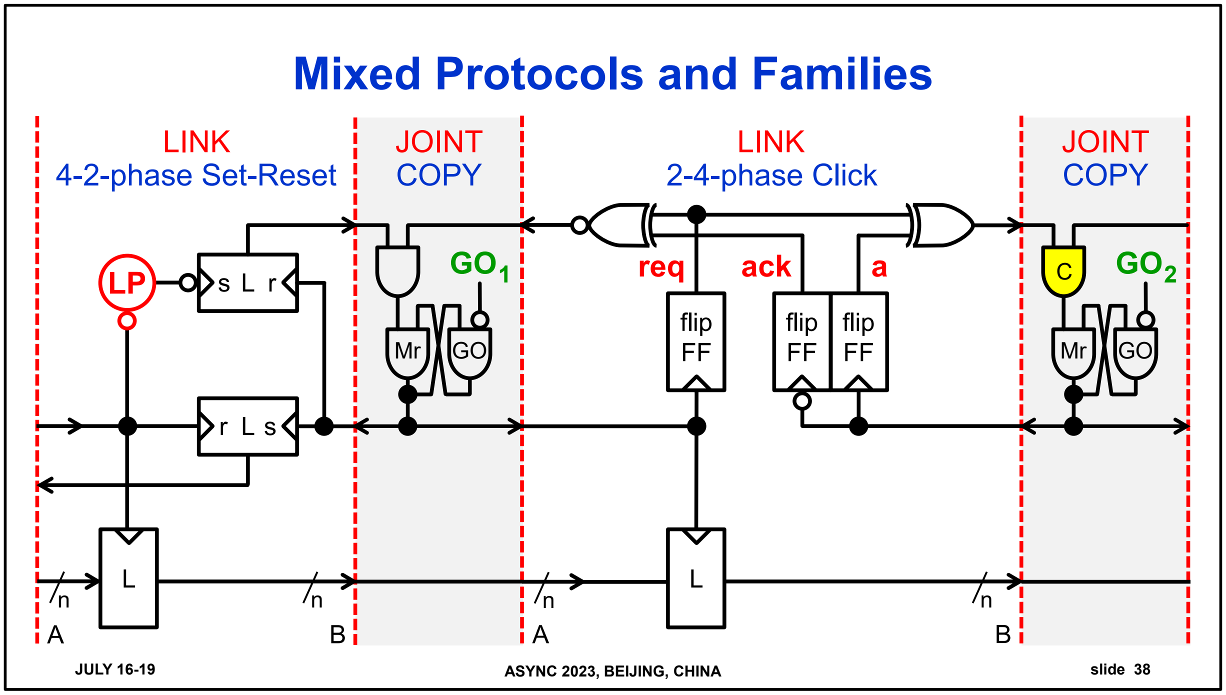
ASYNC 2023, BEIJING, CHINA

slide 37

We can connect the two Joints with a Link that uses

- 2-phase communication at its left port – A
- and 4-phase communication at its right port – B.

This is a Link in the Click circuit family.



Mixed Protocols and Families

JULY 16-19 ASYNC 2023, BEIJING, CHINA slide 38

Also, We can close the loop and create a ring FIFO
with a Link on the left that uses

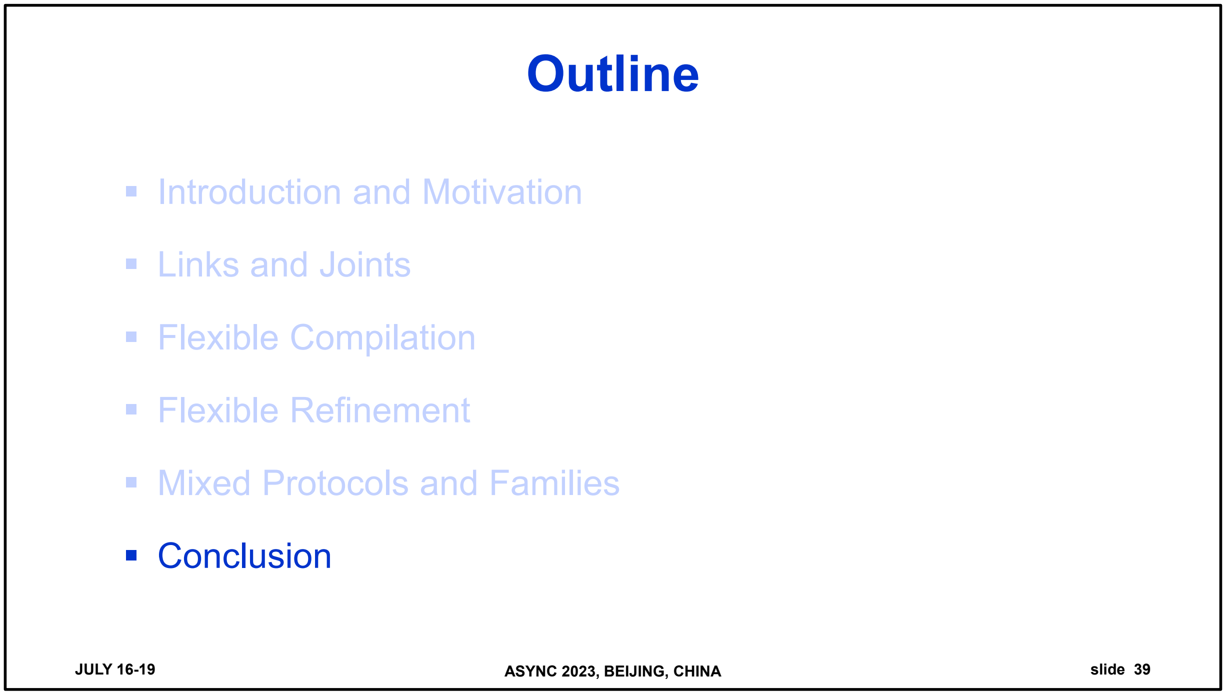
- 4-phase communication at its port A
- and 2-phase at its port B.

The leftmost Link belongs to the Set-Reset family.
<PAUSE for 2 seconds>

Links and Joints make it easy

- to mix and match different protocols and circuit families
- using the same ACT program and Link-Joint network.

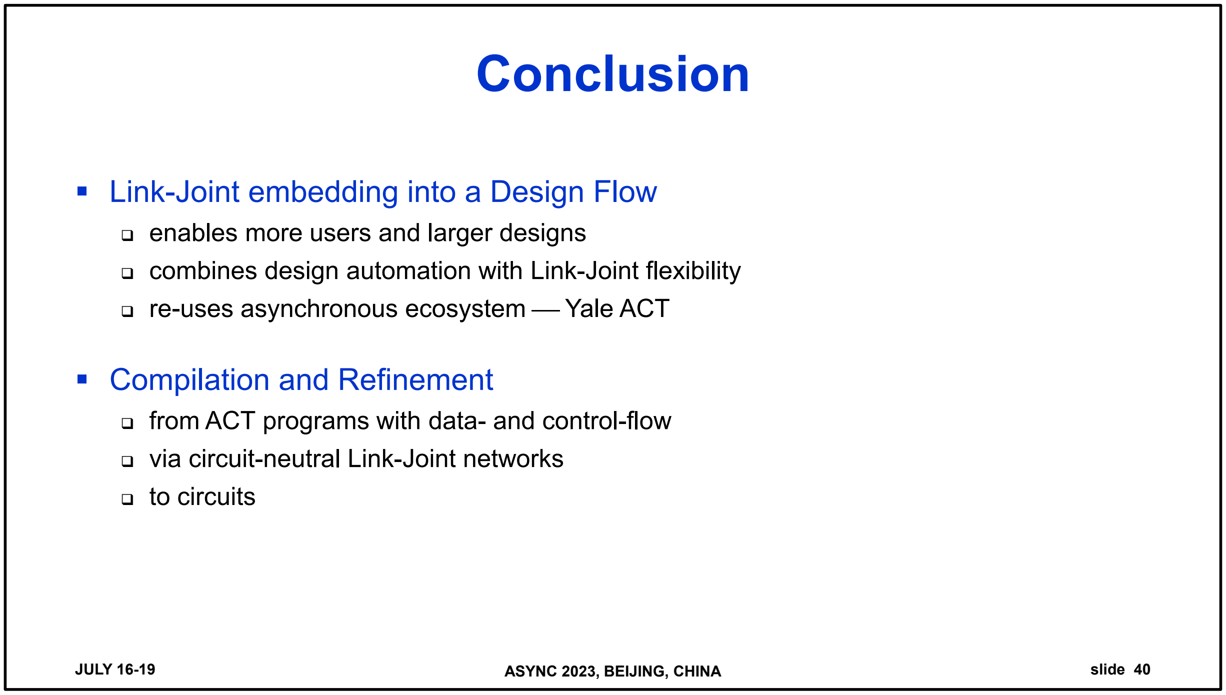
You can read more about mixed protocols in the paper.



Outline

- Introduction and Motivation
- Links and Joints
- Flexible Compilation
- Flexible Refinement
- Mixed Protocols and Families
- Conclusion

To conclude:



Conclusion

- Link-Joint embedding into a Design Flow
  - enables more users and larger designs
  - combines design automation with Link-Joint flexibility
  - re-uses asynchronous ecosystem — Yale ACT
- Compilation and Refinement
  - from ACT programs with data- and control-flow
  - via circuit-neutral Link-Joint networks
  - to circuits

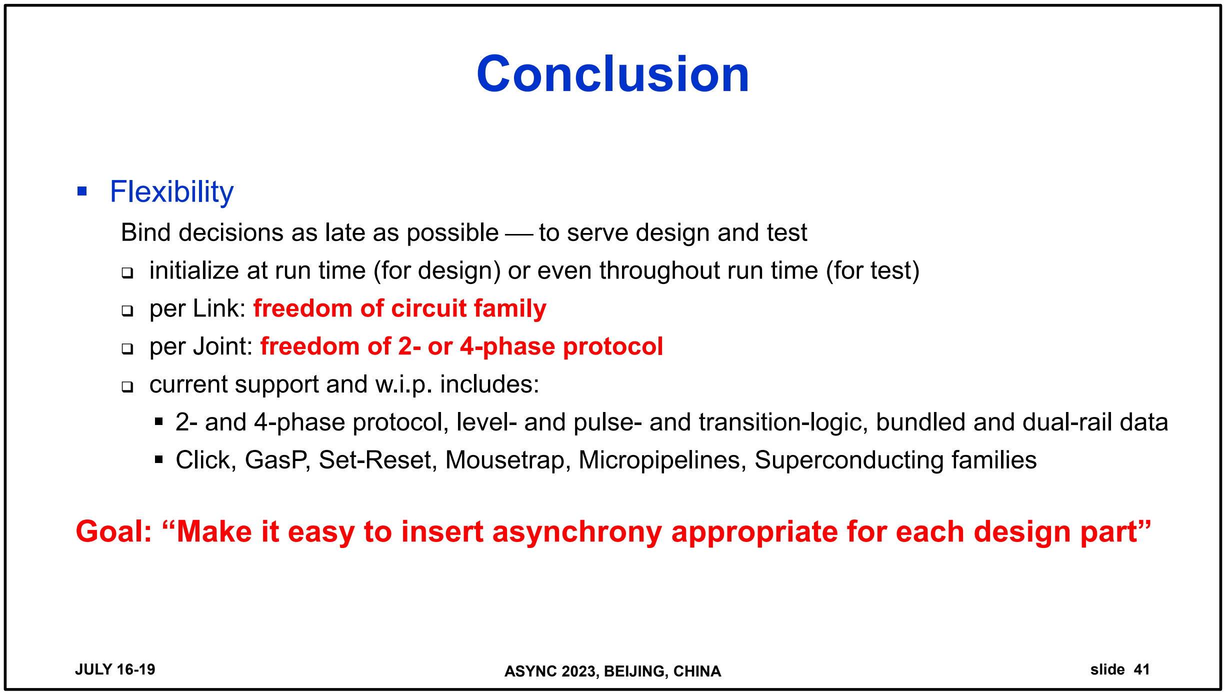
I showed you a Link-Joint embedding into a design flow

- that enables more users and larger designs
- and combines design automation with Link-Joint flexibility

Also, I showed you how compilation and refinement translate

- ACT programs with data- and control-flow
- via circuit-neutral Link-Joint networks into circuits.

I **did not** show you how we model and validate this approach. You can read that in the paper.



Conclusion

- Flexibility

  Bind decisions as late as possible — to serve design and test

  - initialize at run time (for design) or even throughout run time (for test)
  - per Link: **freedom of circuit family**
  - per Joint: **freedom of 2- or 4-phase protocol**
  - current support and w.i.p. includes:
    - 2- and 4-phase protocol, level- and pulse- and transition-logic, bundled and dual-rail data
    - Click, GasP, Set-Reset, Mousetrap, Micropipelines, Superconducting families

Goal: "Make it easy to insert asynchrony appropriate for each design part"

As I said in the beginning of my talk,
Our approach is flexible, because it binds decisions as late as possible.
For example:

- Each Link can choose its circuit family freely, independent of other Links.
- Each Joint can choose freely to use a 2-phase or a 4-phase protocol, independent of other Joints.
- Here is a list of protocols, logic, data encodings and circuit families that we support or are working on.

I want to end with the goal of the Link-Joint approach: <EMPHASIZE SLOWLY>
Make it easy to insert asynchrony appropriate to each design part!

This ends my presentation. I'd be happy to answer questions. Thank you!

THANK YOU!