Design and Test of Asynchronous Systems using the Link and Joint model

Dissertation Defense Presentation
April 25, 2024

Ebele Esimai

Asynchronous Research Center
Department Of Computer Science
Portland State University

Acknowledgements

Advisor: Marly Roncken

Committee:
Dr. Mark P. Jones Dr. Andrew Tolmach
Dr. Xiaoyu Song Dr. Gary Delp

Collaborators:
Rajit Manohar, Yale University
Warren Hunt, University of Texas at Austin
Gary Delp and Mayo Clinic Special Purpose Processor Development Group Team
Ivan Sutherland and Asynchronous Research Center Team

Introduction

Soft Watch at The Moment of First Explosion (Salvador Dali, 1954)

Asynchronous Systems

- ***no*** global, periodic and common clock
- ***no*** global knowledge

but rather:

- local communication, synchronization
- local computation and flow control
- multiple implementation styles
  - Handshake protocol, e.g., 2-phase, 4-phase
  - Data encoding, e.g., bundled, dual-rail
  - Signaling logic, e.g., level, transition, pulse
  - Circuit family, e.g., Click, GasP, Set-Reset, Mousetrap, RSFQ
  - Technology, e.g., CMOS, Superconducting

Focus of Dissertation

Link-Joint network
Embrace the many handshake protocols data storage/coding circuit families, etc.

Goal 1:

Use circuit-neutral Link-Joint networks

- to embrace the many
  - protocols, data encodings, signaling logics
  - circuit families and fabrics
- in one design, test, and debug approach

How?

- hide family differences from interfaces
- facilitate mixing protocol and signaling styles

Benefit:

- clear and uniform design and test interfaces
- flexibility of implementation decisions
- facilitates collaboration and design reuse

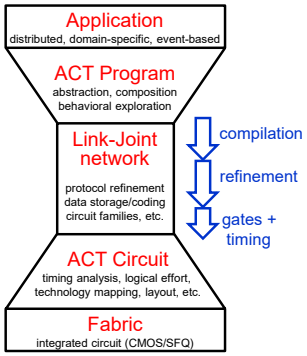
Challenge:

- design by hand limits scaling and users

Focus of Dissertation

Resulting Design-Test Flow:

Ọnà



Goal 2:

Embed Link-Joint into an existing flow

- Yale ACT (Asynchronous Circuit Toolkit)
- to increase scaling and users

How?

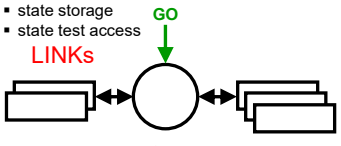
- Shallow embedding as middle layer:
  - for free design and test exploration
  - independent of ACT
  - with full re-use of ACT front and back ends
- Circuit-neutral Compilation:
  - from algorithmic ACT programs
  - to circuit-neutral Link-Joint networks
- Targeted Refinement (stepwise):
  - from Link-Joint networks
  - to ACT circuits

Outline

- Introduction
- Links and Joints
- Design
  - Compilation
  - Refinement
- Test and Debug
  - Uniform Test Approach
- Showcase
  - Mixing Protocols and Families
- Conclusion, Contributions and Future Work

Links and Joints

- communication
- state storage
- state test access



- computation
- flow control
- go-nogo test control

Link-Joint Model:

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

Link-Joint network:

- alternates Links and Joints

Link:

- shares and stores state
- connects two Joints

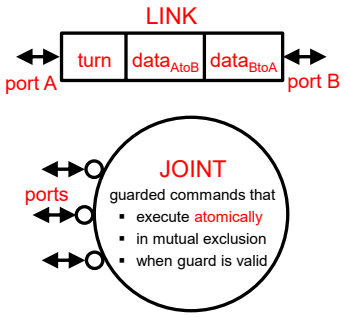
Joint:

- 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

Links and Joints: protocol and model



Protocol:

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

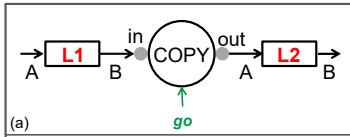
Link:

- 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

Links and Joints: protocol and model



(a)

$myturn(in, out) \wedge go$
$\longrightarrow$
$myW(out) := myR(in)\,;$
$yourturn(in, out)$

(b)

Joint COPY	Link L1	Link L2
myturn(in)	L1.turn == B	
myturn(out)		L2.turn == A
myW(out)		L2.dataAtoB
myR(in)	L1.dataAtoB	
yourturn(in)	L1.turn := A	
yourturn(out)		L2.turn := B

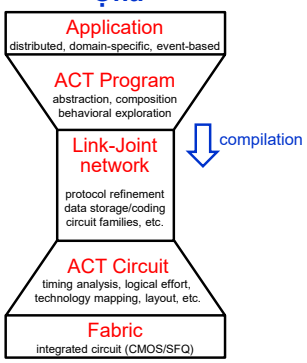
(c)

- terminology:
  - $myturn(p)$: TRUE if p has permission to change the Link state
  - $myR(p)$: Link data that p can read
  - $myW(p)$: Link data that p can write
  - $yourturn(p)$: relinquish permission
  - go : external signal for initialization and test
- atomicity:
  Link states update all at once when the command terminates

Outline

Flexible Compilation

Design-Test Flow:
Ọnà



Strategy:
- Syntax-directed translation
- based on ACT (Asynchronous Circuit Toolkit) compiler

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

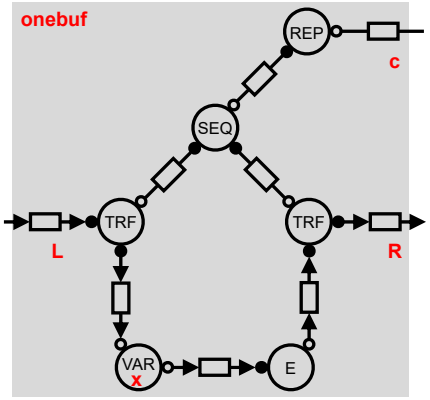
Target: circuit-neutral Link-Joint networks

Challenge:
- ***not*** **compiler**
  - like Philips, Manchester, Caltech, Yale
- ***but*** **Link-Joint library elements** compiled into

Outline

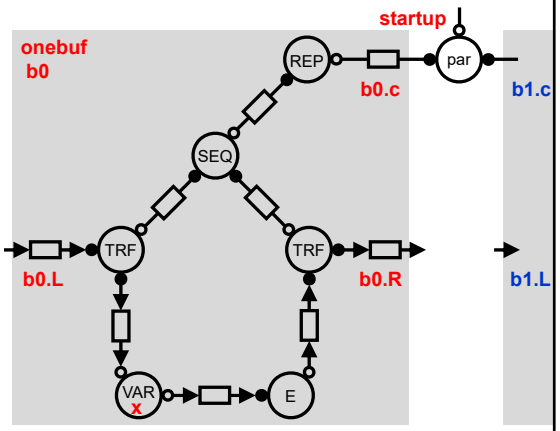
Compilation: CHP

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



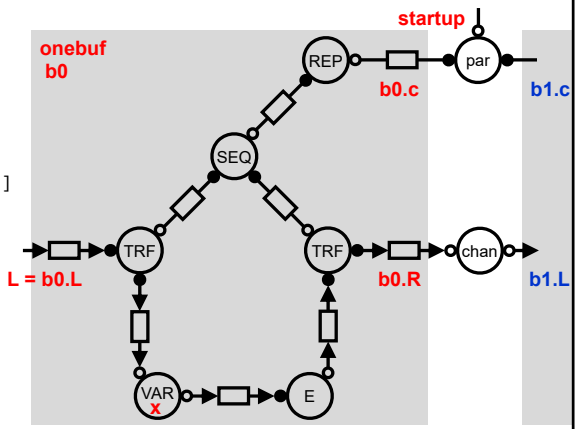
Compilation: CHP

```
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
}
```



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
}
```



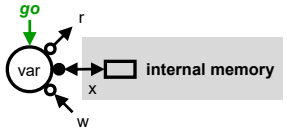
Outline

- Introduction
- Links and Joints
- Design
  - Compilation: Library elements
  - Refinement
- Test and Debug
  - Uniform Test Approach
- Showcase
  - Mixing Protocols and Families
- Conclusion, Contributions and Future Work

Compilation: library elements

Variable icon:

Variable basic Link-Joint network:



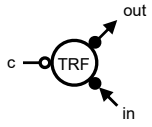
Variable control flow:

Variable guarded command specification:

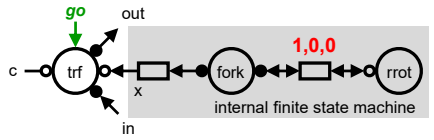
- $myturn(r) \wedge myturn(x) \wedge \boldsymbol{go} \rightarrow$
  $myW(r) := myR(x)\,;\ yourturn(r)$
- $myturn(w) \wedge myturn(x) \wedge \boldsymbol{go} \rightarrow$
  $myW(x) := myR(w)\,;\ yourturn(w)$

Compilation: library elements

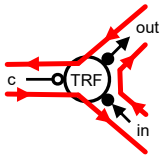
Transfer icon:



Transfer basic Link-Joint network:



Transfer control flow:



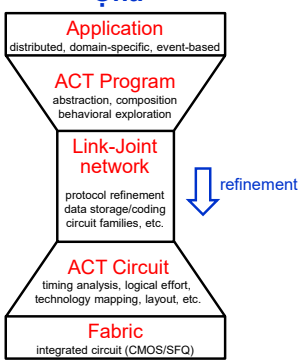
Transfer guarded command specification:

- $myturn(c, in, out, x) \wedge \boldsymbol{go} \wedge myR(x)[0] \rightarrow$
  $yourturn(in, x)$
- $myturn(c, in, out, x) \wedge \boldsymbol{go} \wedge myR(x)[1] \rightarrow$
  $myW(out) := myR(in)\,;\ yourturn(out, x)$
- $myturn(c, in, out, x) \wedge \boldsymbol{go} \wedge myR(x)[2] \rightarrow$
  $yourturn(c, x)$

Outline

Flexible Refinement

Design-Test Flow:
Ọnà



Strategy:
- stepwise decisions for design and test

Source:
- circuit-neutral Link-Joint networks

Target:
- Link-Joint networks
- gate-level circuits

Challenge:
- preserve relation to program

Link-Joint Network Refinement examples:
- **protocol** e.g., 2-phase and 4-phase
- **signaling** e.g., level, transition, and pulse logic
- **data encodings** e.g., bundled and dual-rail data
- **data storage** e.g., where and where not
- **test** e.g., throughput counters, variables to control and observe
- **selection** e.g., arbitrated, priority-ordered, round-robin

Outline

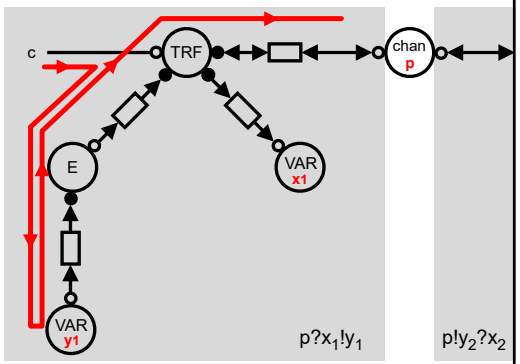
- Introduction
- Links and Joints
- Design
  - Compilation
  - Refinement: Data storage refinement example
- Test and Debug
  - Uniform Test Approach
- Showcase
  - Mixing Protocols and Families
- Conclusion, Contributions and Future Work

Refinement: to store data — or not

ACT program fragments:
- `chp{..p?x1!y1..}`
- `chp{..p!y2?x2..}`
- bidirectional channel: **p**
- variables: **x1, x2, y1, y2**

Path behavior:
- Link stores data for later part
- VAR **y1** stores data for **p**

Goal:
- Avoid data storage in-between



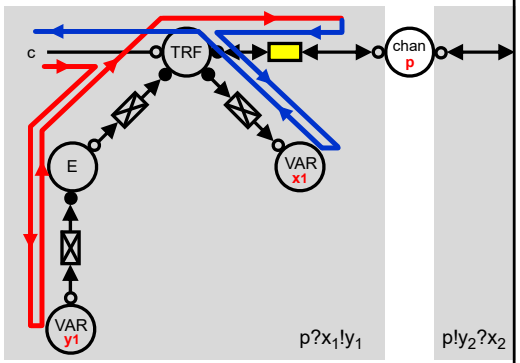
Refinement: to store data — or not

ACT program fragments:
- `chp{..p?x1!y1..}`
- `chp{..p!y2?x2..}`
- bidirectional channel: **p**
- variables: **x1, x2, y1, y2**

Path behavior:
- Link stores data for later part
- VAR **y1** stores data for **p**

Goal:
- Avoid data storage in-between

Solution 1:
- keep internal storage (VAR)
- store data *FROM* chan
- ***no*** storage otherwise

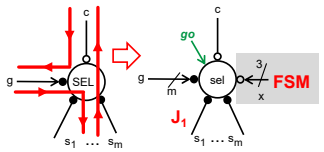


Outline

- Introduction
- Links and Joints
- Design
  - Compilation
  - Refinement: Selection implementation refinement example
- Test and Debug
  - Uniform Test Approach
- Showcase
  - Mixing Protocols and Families
- Conclusion, Contributions and Future Work

Refinement: which selection?

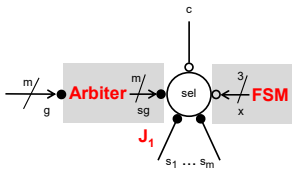
Non-deterministic selection:



guarded command specification:

$myturn(c, g, s_1..s_m, x) \wedge \boldsymbol{go} \rightarrow$
$myR(x)[0] \rightarrow yourturn(g, x)$
$myR(x)[1] \wedge myR(g)[i] \rightarrow yourturn(s_{i+1}, x)$
$myR(x)[2] \rightarrow yourturn(c, x)$

Implementation 1: with arbitration

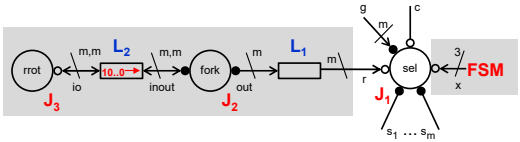


guarded command specification:

$myturn(c, sg, s_1..s_m, x) \wedge \boldsymbol{go} \rightarrow$
$myR(x)[0] \rightarrow yourturn(sg, x)$
$myR(x)[1] \wedge myR(sg)[i] \rightarrow yourturn(s_{i+1}, x)$
$myR(x)[2] \rightarrow yourturn(c, x)$

Refinement: which selection?

Implementation 2: with round-robin



guarded command specification:

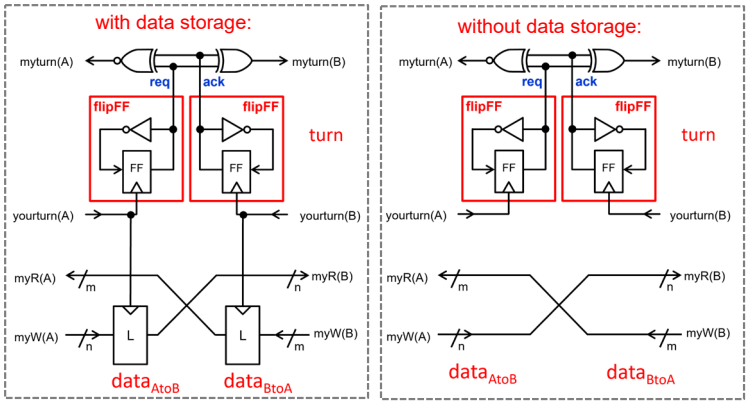
$myturn(c, g, r, s_1..s_m, x) \wedge \boldsymbol{go} \rightarrow$
$myR(x)[0] \rightarrow yourturn(g, x)$
$myR(x)[1] \wedge myR(r)[k] \wedge g[k] \rightarrow yourturn(s_{k+1}, r, x)$
$myR(x)[1] \wedge myR(r)[k] \wedge (g[0] \vee .. \vee g[m-1]) \wedge \neg g[k] \rightarrow yourturn(r)$
$myR(x)[2] \rightarrow yourturn(c, x)$

Outline

Refinement: which circuit family?

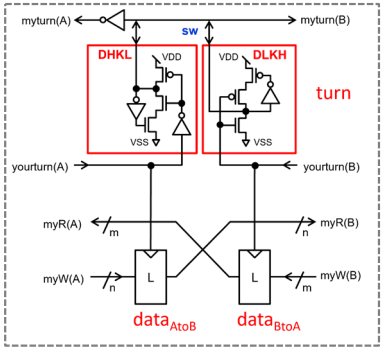
Click Link:

2-phase + bundled data

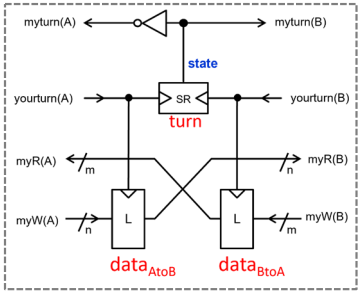


Refinement: which circuit family?

GasP Link:
2-phase + bundled data

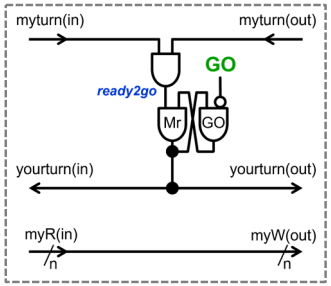


Set-Reset Link:
2-phase + bundled data

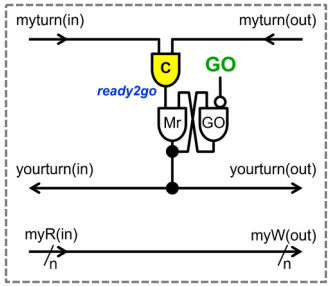


Refinement: which protocol?

COPY Joint:
2-phase protocol



COPY Joint:
4-phase protocol



Outline

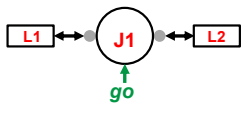
- Introduction
- Links and Joints
- Design
  - Compilation
  - Refinement
- Test and Debug
  - Uniform Test Approach
- Showcase
  - Mixing Protocols and Families
- Conclusion, Contributions and Future Work

Test and Debug

```
defproc FIFO(){
 chan(int<8>) b1; chan(int<8>) b2;
 chan(int<8>) b3; ...
 dataflow { b1 -> b2; b2 -> b3;
            b3 -> b4;    ...
 }
}
```

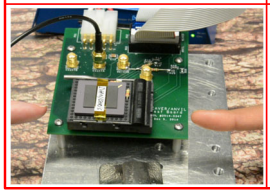
To test algorithmic programs
- so many lines – so few exports

use
- interactive code debug
- to read/write states and set breakpoints
- for single- and multi-step tests



To test Link-Joint networks
- so many Links, Joints - so few external ports

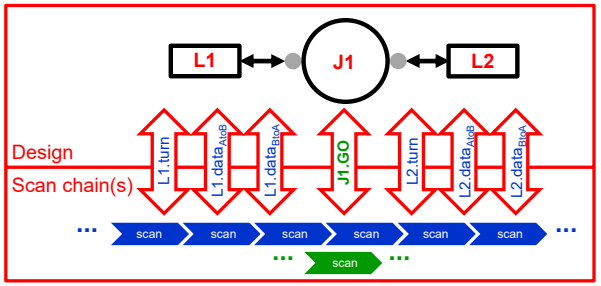
use
- Link variables and Joint go signals
- to access states and enable/disable actions
- for single-step and multi-step tests



To test hardware
- so many wires – so few pins

use
- scan
  - to read/write states and go
  - with a small number of pins
- MrGO to permit and prohibit actions
- for single- and multi-step tests

Test and Debug



Existing:

- 1-to-1 relation between Link-Joint states/go and circuit-level scan

Goal:

- Extend this relation to the ACT program level
- Develop tests at the program level, and translate down

Outline

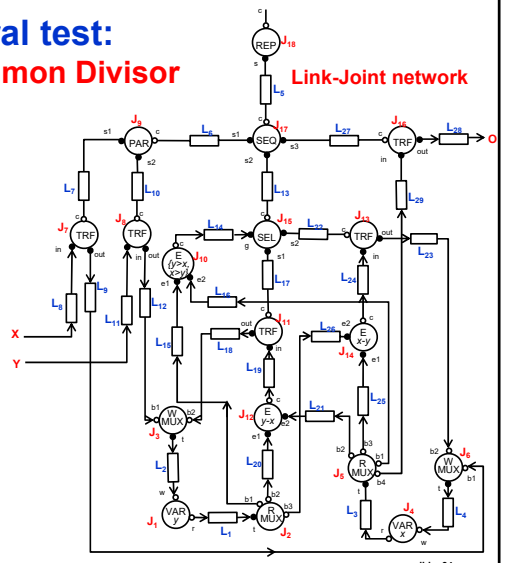
- Introduction
- Links and Joints
- Design
  - Compilation
  - Refinement
- Test and Debug
  - Uniform Test Approach: Structural Test example
- Showcase
  - Mixing Protocols and Families
- Conclusion, Contributions and Future Work

Structural test: Greatest Common Divisor

Program

```
defproc gcd2
(chan?(int) X,Y; chan!(int) O)
 {
  int x, y;
  chp {
    *[pc1:  X?x,  Y?y;
      pc2: *[y > x ->
              log("Guard 1");
              pc3: y := y - x
            [] x > y ->
              log("Guard 2");
              pc4: x := x - y
            ];
      log("Out of loop");
      pc5: O!x
    ]
  }
}
```

Link-Joint network



Structural test: Greatest Common Divisor

Program

```
defproc gcd2
(chan?(int) X,Y; chan!(int) O)
 {
  int x, y;
  chp {
    *[pc1:  X?x,  Y?y;
      pc2: *[y > x ->
              log("Guard 1");
              pc3: y := y - x
            [] x > y ->
              log("Guard 2");
              pc4: x := x - y
            ];
      log("Out of loop");
      pc5: O!x
    ]
  }
}
```

Simulated with **ACTSIM**,
a language simulator in the Yale ACT tool suite

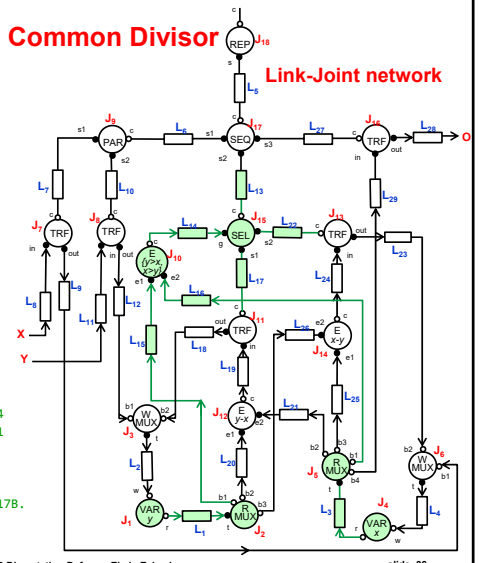
ACTSIM Script	ACTSIM Output
1. watch x y	[0] <[env]> y := 25 (0x19)
2. breakpt x	[0] <[env]> x := 7 (0x7)
3. breakpt y	[0] <> Guard 1 chosen
4. goto pc2	[10] <> y := 18 (0x12)
5. set y 25	[10] <> *** breakpoint y
6. set x 7	[10] <[env]> y := 7 (0x7)
7. cycle	[10] <[env]> x := 25 (0x19)
8. goto pc2	[10] <> Guard 2 chosen
9. set y 7	[20] <> x := 18 (0x12)
10. set x 25	[20] <> *** breakpoint x
11. cycle	[30] <> Guard 2 chosen
12. cycle	[40] <> x := 11 (0xB)
13. goto pc2	[40] <> *** breakpoint x
14. set y 7	[40] <[env]> x := 7 (0x7)
15. set x 7	[40] <> Out of loop
16. cycle	

Structural test: Greatest Common Divisor

Verilog testbench

```
1.  initial begin
2.  $dumpvars();

3.  // Guard 1: y > x
4.  // Disable all Joint actions.
5.  go[18:1] = {18{1'b0}}; #10

6.  // Goto pc2, which is L13B.
7.  L13.reg_Amyturn = 0;
8.  L13.reg_Bmyturn = 1;
9.  // Set VAR y & VAR x to guard 1 true
10. J1.st_ABin = 25;  // Var y
11. J4.st_ABin = 7;  // Var x

12. // Enable Joints for guard selection
13. go[15]  = 1'b1;  // SEL J15
14. go[10]  = 1'b1;  // E J10
15. go[5:4] = 2'b11; // RMUX x J5, VAR x J4
16. go[2:1] = 2'b11; // RMUX y J2, VAR y J1

17. // Run long enough
18. #220
19. // Check selection of statement 1 at L17B.
...
```

Link-Joint network



Connecting test at all abstraction levels

Control-Observation		Program	Link-Joint network
Guard 1	start point	goto pc2	L13.reg_Amyturn = 0; L13.reg_Bmyturn = 1;
	start values	set y 25 set x 7	J1.st_ABin = 25; J4.st_ABin = 7;
	stop point	breakpt y	J11
	stop observation	"Guard 1 is chosen"	statement 1 at L17A
Guard 2	start point	goto pc2	L13.reg_Amyturn = 0; L13.reg_Bmyturn = 1;
	start values	set y 7 set x 25	J1.st_ABin = 7; J4.st_ABin = 25;
	stop point	breakpt x	J13
	stop observation	"Guard 2 is chosen"	statement 2 at L22A
Exit	start point	goto pc2	L13.reg_Amyturn = 0; L13.reg_Bmyturn = 1;
	start values	set y 7 set x 7	J1.st_ABin = 7; J4.st_ABin = 7;
	stop point	breakpt x	J17
	stop observation	"Out of loop"	startup at L13A

Hand over key information

- **start:**
  - where does the operation start
  - what are the (key) initial values
- **stop:**
  - where does the operation stop
  - what are the (key) end values

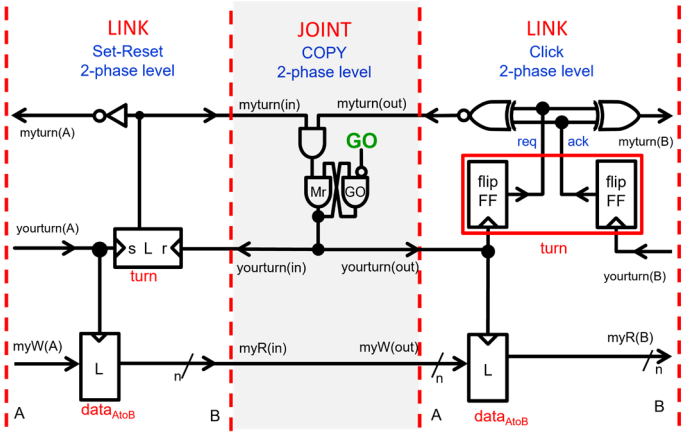
Expand values later

- **start-stop values:**
  - come from test pattern generation
  - based on hardware fault models
- **can be detailed later**
  - for hardware fault coverage

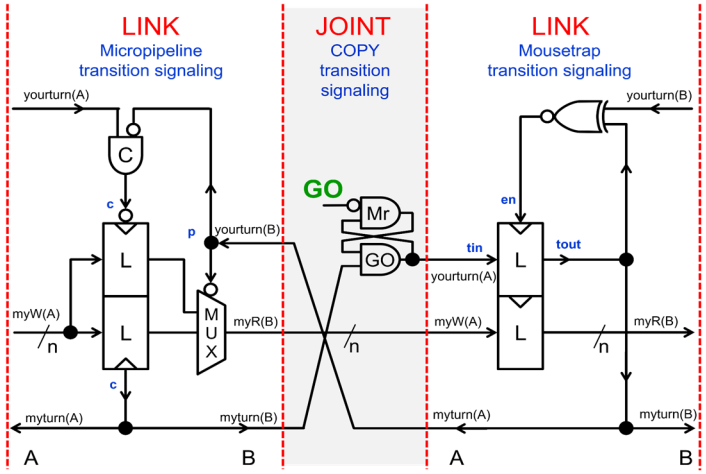
Outline

- Introduction
- Links and Joints
- Design
  - Compilation
  - Refinement
- Test and Debug
  - Uniform Test Approach
- Showcase
  - Mixing Protocols and Families
- Conclusion, Contributions and Future Work

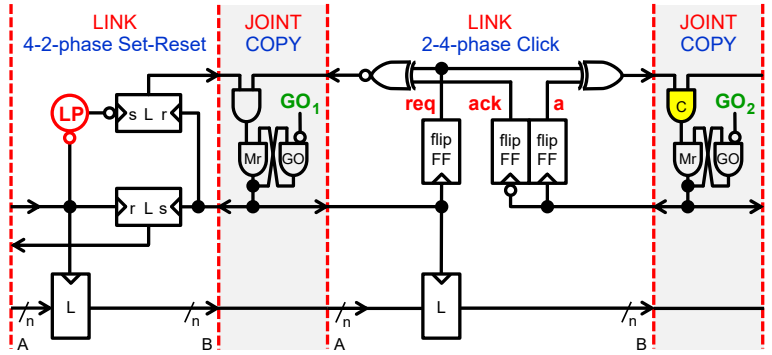
Mixing Circuit Families



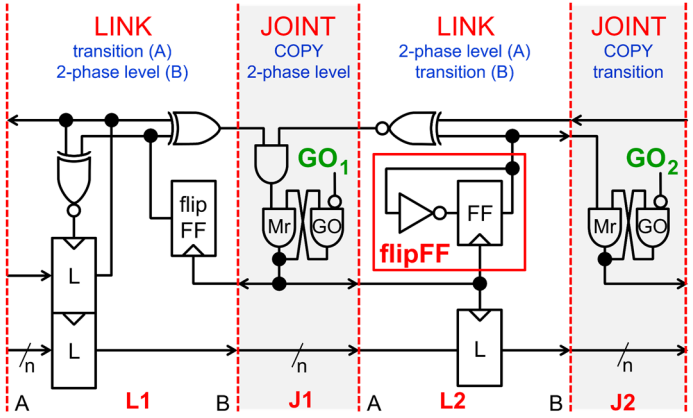
Mixing Circuit Families



Mixing Protocols



Mixing Signaling Logics



Outline

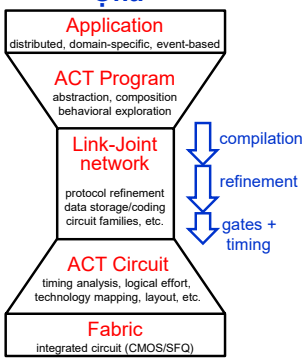
- Introduction
- Links and Joints
- Design
  - Compilation
  - Refinement
- Test and Debug
  - Uniform Test Approach
- Showcase
  - Mixing Protocols and Families
- Conclusion, Contributions and Future Work

Conclusion

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

Resulting Design-Test Flow:

Ọnà



Created: Ọnà

- Design and test flow for asynchronous systems
- based on:
  - Link-Joint model — as middle layer
  - Yale ACT design flow — above and below
- to:
  - embrace the many asynchronous styles
  - scale applications and users

Benefits:

- General and flexible design, test, and debug
- Supportive of collaboration

Implementation styles investigated include:

- 2- and 4-phase protocols
- bundled and dual-rail data encodings
- level- and pulse- and transition- signaling logics
- Click, GasP, Set-Reset, Mousetrap, Micropipelines and Superconducting families

Future Work

- Deep embedding of Links and Joints into Yale's Asynchronous Circuit Toolkit (ACT)
- Compiler optimizations for Communicating Hardware Processes (CHP) programs (work in progress at Yale)
- Comprehensive test extension into hardware test coverage
- Model equivalence between abstraction levels

My Contributions

- ❑ Link-Joint shared variable semantics and Link-Joint port connection
- ❑ Compilation of ACT programs into Link-Joint networks
  - Adaption of Yale's CHP2PRS compiler to generate Link-Joint networks
- ❑ Implementation of Link-Joint network refinements
- ❑ Unified test and debug translation between abstraction levels
  - Extension of existing relation from Links and Joints, and circuit scan to programs
  - New commands for ACTSIM: *skip-comm, gc-retry, goto*
- ❑ Identified that initialization determines active-passive/push-pull protocol settings
  - Link variable *turn* makes old notions for active-passive and push-pull irrelevant
- ❑ Verilog behavioral modules for Links and Joints
  - Implementation and validation of Link-Joint networks and their refinements
  - Python program to automate generation of the Verilog modules and associated testbench

Publications

Ebelechukwu Esimai and Marly Roncken
Flexible Compilation and Refinement of Asynchronous Circuits,
2023 28th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC), Beijing, China, 2023, pp. 109-119

Marly Roncken, Ebelechukwu Esimai, Vivek Ramanathan, Warren A. Hunt and Ivan Sutherland
State Access for RSFQ Test and Analysis,
IEEE Transactions on Applied Superconductivity, vol. 33, no. 5, pp. 1-7, Aug. 2023

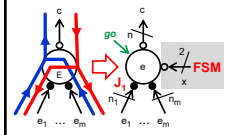
Ebelechukwu Esimai and Marly Roncken
Flexible Active-Passive and Push-Pull Protocols,
IEEE Embedded Systems Letters, vol. 14, no. 3, pp. 139-142, Sept. 2022

Marly Roncken, Ivan Sutherland, and Ebelechukwu Esimai,
Micropipelines United, in A. Brown and A. Yakovlev (eds)
We're going to Need a Bigger Computer - Essays dedicated to Steve Furber on the occasion of his retirement. At Last, University of Manchester Press Unit, 12 January 2024.

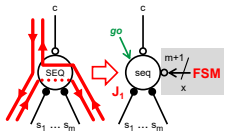
ASYNC 2022 Summer School: 3-day online seminar
Rajit Manohar (Yale University), Benjamin Hill (Intel), Montek Singh (University of North Carolina at Chapel Hill), Marly Roncken, Ebelechukwu Esimai, and Ivan Sutherland (Portland State University).
The Portland State presentations cover: Links and Joints (1) behavioral design (2) gate-level design.

THANK YOU!

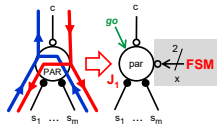
Compilation: library elements



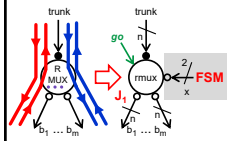
$myturn(c, e_1..e_m, x) \wedge \ go \wedge myR(x)[0] \rightarrow$
$yourturn(e_1..e_m, x)$
$myturn(c, e_1..e_m, x) \wedge \ go \wedge myR(x)[1] \rightarrow$
$myW(c) := f(myR(e_1..e_m));$
$yourturn(c, x)$



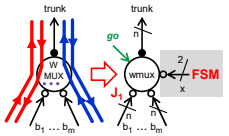
$myturn(c, s_1..s_m, x) \wedge \ go \wedge myR(x)[i] \rightarrow$
$yourturn(s_{i+1}, x)$
$myturn(c, s_1..s_m, x) \wedge \ go \wedge myR(x)[m] \rightarrow$
$yourturn(c, x)$



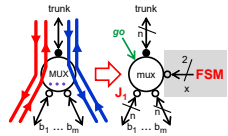
$myturn(c, s_1..s_m, x) \wedge \ go \wedge myR(x)[0] \rightarrow$
$yourturn(s_1..s_m, x)$
$myturn(c, s_1..s_m, x) \wedge \ go \wedge myR(x)[1] \rightarrow$
$yourturn(c, x)$



$myturn(b_i, trunk, x) \wedge \ go \wedge myR(x)[0] \rightarrow$
$yourturn(trunk, x)$
$myturn(b_i, trunk, x) \wedge \ go \wedge myR(x)[1] \rightarrow$
$myW(b_i) := myR(trunk);$
$yourturn(b_i, x)$

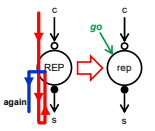


$myturn(b_i, trunk, x) \wedge \ go \wedge myR(x)[0] \rightarrow$
$myW(trunk) := myR(b_i);$
$yourturn(trunk, x)$
$myturn(b_i, trunk, x) \wedge \ go \wedge myR(x)[1] \rightarrow$
$yourturn(b_i, x)$

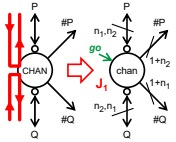


$myturn(b_i, trunk, x) \wedge \ go \wedge myR(x)[0] \rightarrow$
$myW(trunk) := myR(b_i);$
$yourturn(trunk, x)$
$myturn(b_i, trunk, x) \wedge \ go \wedge myR(x)[1] \rightarrow$
$myW(b_i) := myR(trunk);$
$yourturn(b_i, x)$

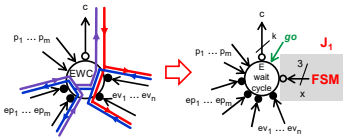
Compilation: library elements



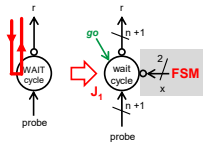
$myturn(c, s) \wedge \ go \rightarrow$
$yourturn(s)$



$myturn(P, Q) \wedge \ go \rightarrow$
$myW(P) := myR(Q); myW(Q) := myR(P);$
$yourturn(trunk, x)$
$\#P \equiv \{myR(Q), myturn(Q)\}$
$\#Q \equiv \{myR(P), myturn(P)\}$



\\ *evaluate non-probe expressions (get latest values of the variables)*
$myturn(c, ev_1..ev_m, ep_1..ep_m, x) \wedge \ go \wedge myR(x)[0]$
$\rightarrow yourturn(ev_1..ev_n, x)$
\\ *wait until some guard is TRUE before taking probe snapshots*
$myturn(c, e_1..e_n, ep_1..ep_m, x) \wedge \ go \wedge myR(x)[1] \ \wedge (f'[1] \vee ..\vee f'[k])$
$\rightarrow yourturn(ep_1..ep_m, x)$
\\ *return stable evaluation results*
$myturn(c, e_1..e_n, ep_1..ep_m, x) \wedge \ go \wedge myR(x)[2]$
$\rightarrow myW(c) := \{f[1]..f[k]\}; \ yourturn(c, x)$



\\ *arbitrate for one FSM cycle*
$myturn(r, x) \wedge \ go \wedge myR(x)[0] \ \rightarrow yourturn(x)$
\\ *return stable probe snapshot*
$myturn(r, x) \wedge \ go \wedge myR(x)[1]$
$\rightarrow myW(r) := (0 \ if \ grant_x, probe \ if \ grant_{probe}); \ yourturn(r, x)$